A Comparison of Serial & Parallel Particle Filters for Time Series Analysis

by

David Klemish

Departments of Statistical Science & Economics
Duke University

Date: _____

Approved:

_____
Juan Rubio-Ramirez, Supervisor

_____
Mike West

_____
Charles Becker

Thesis submitted in partial fulfillment of the
requirements for the degree of
Master of Science in the Departments of Statistical Science & Economics
in the Graduate School of Duke University
2014

Abstract

A Comparison of Serial & Parallel Particle Filters for Time Series Analysis

by

David Klemish

Department of Statistical Science & Economics
Duke University

Date: _____

Approved:

_____
Juan Rubio-Ramirez, Supervisor

_____
Mike West

_____
Charles Becker

An abstract of a thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in the Departments of Statistical Science & Economics
in the Graduate School of Duke University
2014

Abstract

This paper discusses the application of parallel programming techniques to the estimation of hidden Markov models via the use of a particle filter. It highlights how the Thrust parallel programming language can be used to implement a particle filter in parallel. The impact of a parallel particle filter on the running times of three different models is investigated. For particle filters using a large number of particles, Thrust provides a speed-up of five to ten times over a serial C++ implementation, which is less than reported in other research.

Contents

List of Tables

List of Figures

1

Introduction

Modern statistical and econometric techniques are heavily dependent on computational processing to solve increasingly difficult problems. Advances in computer hardware bring the promise of solving existing problems faster and bring solutions to previously intractable problems within reach. However, each new generation of hardware requires researchers to spend time and effort to learn new programming skills to take advantage of these advances, even when programming is not their main interest. This paper tries to quantify the impact of using graphical processing units, a low-cost coprocessor that enables easier parallel programming, on a class of statistical models so that others can judge whether the benefit of learning the new skill required is worth the effort required.

The remainder of this paper is organized as follows. Chapter 2 presents a brief discussion of the use particle filters to derive a filtering distribution for state variables in a hidden Markov model and Chapter 3 introduces the basics of parallel programming for GPUs, focusing on the CUDA and Thrust programming languages. Chapter 4

through 6 all discuss the impacts of using a parallel particle filter on the running times of three different hidden Markov models. These models include a simple first-order autoregressive time series with known parameters and Gaussian distributed observational error in chapter 4, the same model with unknown parameters in chapter 5, and a model with non-Gaussian distributed errors with time-varying parameters in chapter 6. Chapter 7 provides a conclusion.

2

Particle Filters & Parallel Processing

2.1 Hidden Markov Models

Hidden Markov models (HMMs) are a class of statistical models where a discrete time sequence of interest $\mathbf{x}_{1:t}$ follows a first order Markov process where the probability distribution $p(\mathbf{x}_t)$ depends only on the value of $\mathbf{x}_{t-1}$ and potentially other parameters. Conditional on knowing the value of $\mathbf{x}_{t-1}$, $p(\mathbf{x}_t)$ is independent of $\mathbf{x}_i$ for all $i < (t-1)$. The conditional distribution $p(\mathbf{x}_t|\mathbf{x}_{t-1})$ is referred to as the transition distribution.

As part of this model, the realized values of $\mathbf{x}_{1:t}$ are assumed to be unobservable. Instead, at each time period t a value $\mathbf{y}_t$ is observed as a realization from the probability distribution $p(\mathbf{y}_t|\mathbf{x}_t)$. This distribution is referred to as the observation or emission distribution. Conditional on knowing the value of $\mathbf{x}_t$, $\mathbf{y}_t$ is independent of all other $\mathbf{x}_i, i \neq t$ and $\mathbf{y}_j, j \neq t$. Note that $\mathbf{x}_t$ may depend on a transformation of of $\mathbf{x}_{t-1}$ via a function f; similarly $\mathbf{y}_t$ may depend on a transformation of $\mathbf{x}_t$ via

a function g. For notational simplicity, the function notation is suppressed in the below discussion.

The states $\mathbf{x}_t$ and $\mathbf{y}_t$ can be discrete or continuous; mixed models where one state sequence is discrete while the other is continuous are also possible. In the case where $\mathbf{x}_t$ is discrete, the transition density is a transition matrix, where each row sums to one. If $\mathbf{y}_t$ is also discrete, then the observation density is also a matrix. Furthermore, states can also either be scalar or vector valued; for the remainder of the paper only scalar valued sequences are considered.

The joint distribution of all states is

$$\begin{aligned} p(x_{0:t}, y_{1:t}) &= p(x_{0:t})p(y_{1:t}|x_{1:t}) \\ &= p(x_0)\prod_{t=1}^{T} p(x_t|x_{t-1})\prod_{t=1}^{T} p(y_t|x_t) \end{aligned} \tag{2.1}$$

where the second line follows by using the conditional dependence structure inherent in the model specification. Note that it is necessary to start the Markov chain for the hidden states from some initial unobserved state x_0. Assuming the the series x_t is stationary, then the marginal distribution of x_t is given by $\int p(x_t|x_{t-1})p(x_{t-1})dx_{t-1}$ for all t. It is commonly assumed that the distribution for the unobserved state x_0 follows this marginal stationary distribution.

This model can be represented as a directed acyclic graph (DAG) which visually displays the conditional relationships of all variables. An example is displayed in Figure 2.1, omitting any parameters that also enter into the transition and observation densities.

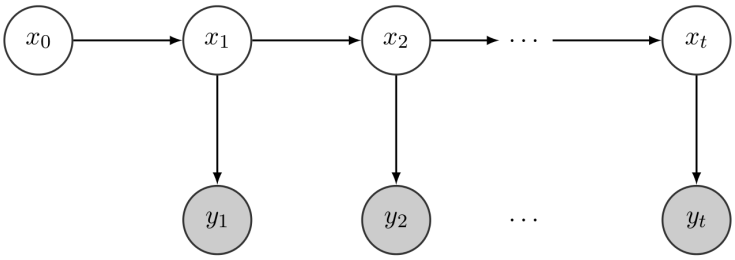


FIGURE 2.1: Graphical model representation of a hidden Markov model

Each node in the graph represents a state in the model, where shaded nodes indicate that the corresponding state has been observed. Arrows indicate conditional dependencies in the joint distribution of all variables in the model. For example, the distribution of x_1 depends on the value x_0 and as a result there is an arrow from the x_0 node to the x_1 node.

Hidden Markov models are used in a variety of applications, including speech recognition (Jelinek, 1997), bioinformatics (Karplus et al., 1998; Krogh et al., 2001), and time series analysis (González et al., 2005; Rydén et al., 1998). Of common interest to all of these applications is making inference on the hidden states $x_{1:t}$ conditional on the observed data $y_{1:t}$. We focus on time series applications in state space models (SSMs), which are HMMs where x_t and y_t are continuous valued.

Note that we also focus on situations where the information set at time t includes only observations up to and including time t, so the marginal conditional distribution of interest for a given time t is $p(x_t|y_{1:t})$. This is in contrast to situations where the information set at time t includes *all* observations from time 1 to $T, t < T$, for which the conditional distribution is $p(x_t|y_{1:T})$. The former is known as a filtering distribution, while the latter is a smoothing distribution.

If the functions f and g are linear functions, the transition and observation den-

sities are distributed Gaussian and all parameters other than the hidden states are known, exact inference on the hidden states is given by the Kalman filter (Kalman, 1960). In cases where the functions f or g are non-linear, the extended Kalman filter and the unscented Kalman filter (Julier and Uhlmann, 1997) can be used for approximate inference. The extended Kalman filter works by linearizing the functions f and g via a Taylor series expansion and then applying the standard Kalman filter. As an alternative to linearizing, the unscented Kalman filter passes a selected set of points through f and g and fits Gaussians to the resulting transformed points. In cases where the transition and observation densities are not Gaussian, the particle filter (Gordon et al., 1993) can be used to approximate the filtering distribution of the hidden states.

2.2 Particle Filters

The particle filter is based on *particles*, which are independent and identically distributed samples $x_{0:t}^i$ for $i = 1, \ldots, N$. If it was possible to sample directly from the distribution $p(x_{0:t}|y_{1:t})$, the distribution could be approximated by

$$\frac{1}{N} \sum_{i=1}^{N} \delta_{x_{0:t}^i}(x_{0:t}) \tag{2.2}$$

However, it is generally impossible to sample directly from this distribution, even if the value of the distribution can be calculated for a given sample, due to the high dimensionality of the distribution. The alternative is to instead sample from a more tractable distribution $q(x_{0:t}|y_{1:t})$, and weight the resulting samples to better approximate the actual distribution $p(x_{0:t}|y_{1:t})$, in a process known as importance

sampling (Doucet, 2001).

2.2.1 Sequential Importance Sampling

The unnormalized weights are

$$w_t^i \propto \frac{p(x_{1:t}^i|y_{1:t})}{q(x_{1:t}^i|y_{1:t})} \tag{2.3}$$

Intuitively, if a sample is relatively more likely under the distribution p than under the distribution q, that sample should be given more weight to better approximate p. These weights are then normalized to sum to unity by dividing each weight by the sum of all weights:

$$\tilde{w}_t^i = \frac{w_t^i}{\sum_i w_t^i} \tag{2.4}$$

The standard approach is to use a sampling distribution $q(x_{1:t}^i|y_{1:t})$ that allows a hidden state x_t to be appended to a previously sampled sequence $x_{0:t-1}$ such that the resulting sequence $x_{0:t}$ is a proper sample from $q(x_{1:t}^i|y_{1:t})$, hence the name sequential importance sampling. In that case, the distribution q can be rewritten as

$$q(x_{0:t}|y_{1:t}) = q(x_t|x_{0:t-1}, y_{1:t})q(x_{0:t-1}|y_{1:t-1}) \tag{2.5}$$

It can be shown (Murphy, 2012) under this assumption that the weights are equivalent to

$$w_t^i = w_{t-1}^i \frac{p(y_t|x_t^i)p(x_t^i|x_{t-1}^i)}{q(x_t^i|x_{0:t-1}^i, y_{1:t})} \tag{2.6}$$

Two other simplifying assumptions commonly used for the sampling distribution q. The first is that the x_t depends only on x_{t-1} and is independent on the history $x_{0:t-2}$

conditional on x_{t-1}. Given this, the second assumption is to use the conditional prior distribution for $p(x_t|x_{t-1})$ as the sampling distribution $q(x_t|x_{t-1}, y_t)$. The result is that the particle weight simplifies to

$$w_t^i = w_{t-1}^i \, p(y_t|x_t^i) \tag{2.7}$$

As a result, a particle filter can be implemented by first initializing a set of particles at time $t = 0$ by drawing from the marginal distribution $p(x_t)$. Then for each subsequent time period, sample a new particle location x_t^i for each particle conditional on its previous location x_{t-1}^i and update that particle's normalized weight.

However, in practice this approach fails, as the weight of one particle tends to unity, while the weight of all other particles goes to zero after only a few time periods (Doucet, 2001). Since this procedure is sampling in a high-dimensional space that increases with the number of time periods, one sample history is inevitably more likely than all others unless an exponentially increasing number of particles is used. The solution to this problem is to eliminate particles with low weight by reassigning the locations of those particles with the locations of high weight particles via resampling.

2.2.2 Sequential Importance Resampling

The sequential importance resampling approach modifies the basic sequential importance sampling method by resampling particle locations if the approximate effective number of particles given by

$$\hat{S}_{eff,t} = \frac{1}{\sum_i \tilde{w}_t^i} \tag{2.8}$$

drops below some selected limit. While there are different possible resampling approaches, the most common is to sample from a multinomial distribution with weights equal to $\tilde{w}_t^i, i = 1, \ldots, N$. Therefore, if particle x_t^j has weight 0.50, each particle would have a 50% chance of being assigned x_t^j's location after resampling. After resampling occurs, the weights at time t are reset to be uniform across particles.

Therefore, the algorithm for the Sequential Importance Resampling (SIR) particle filter can be summarized as:

SIR Particle Filter

1. Let $t = 0$.
2. For each particle i, for $i = 1, \ldots, N$, sample a value x_0^i from the marginal distribution $p(x_0)$.
3. For time $t = 1, \ldots, T$ do:
   (a) For each particle i, sample a value x_t^i for each particle from the conditional distribution $p(x_t|x_{t-1})$.
   (b) For each particle i, calculate the weight w_t^i equal to $w_{t-1}^i p(y_t|x_t^i)$.
   (c) Calculate the total sum of weights $W_t = \sum_{i=1}^N w_t^i$.
   (d) Normalize the weight for each particle by dividing w_t^i by W_t for $\forall i$.
   (e) Store each particle's location and weight.
   (f) Calculate the effective sample size. If it is below the selected limit:
      i. Resample all particle locations based on current normalized weights.
      ii. Reset all weights to $1/N$.

Upon completion, at each time t one can construct the marginal distribution of the hidden state x_t, $p(x_t|y_{1:t})$ from the particle locations and weights for that period. As the number of particles goes to infinity, this approximation will approach the true posterior (Crisan et al., 1998).

However, since every iteration in time in the algorithm requires multiple operations on each particle, actually running this algorithm can be costly in terms of time and computing cost. Since each particle is independent of the others, steps (2), (3a-b), (3d) and (3f.i) can all be performed simultaneously given a computing environment that is capable of parallel computation. Therefore, significant improvements in running times should be achievable given a parallel computer. This would then allow for the use of more particles giving a better approximation to the true posterior distribution of hidden states.

3

Parallel Computation via CUDA & Thrust

3.1 Introduction

Graphical processing units (GPUs) are specialized numerical processors that were developed in the 1990s to enable improved two- and three- dimensional graphics in computer games. These applications required many identical arithmetic operations to be applied to each pixel on the computer screen. Traditional central processing units (CPUs) would be required to loop through the content of each pixel stored in memory before the screen could be updated, which was a slow process which limited the graphics that could be achieved.

As a result, graphics card manufacturers such as NVIDIA and ATI, developed GPUs that consisted of many individual processing units that could do arithmetic calculations simultaneously to each element of a data array, thereby accelerating the process of screen updating. Furthermore, since these GPUs were intended to be sold

to the general public, these chips were sold at fairly low costs. As a trade-off to these advantages, GPUs were designed with less transistors dedicated to logic and memory.

Starting in the early 2000's, researchers realized that the problems they were dealing with could be solved using parallel computation. As a result, they began to directly program GPUs to do numerical scientific computing instead of graphical processing. However, specialized knowledge of graphics card programming was necessary to take advantage of the GPU's inherently parallel architecture, which few researchers possessed.

Graphics card manufacturers realized that scientific computing could be a new market for their products. Therefore, manufacturers began to develop tools that would make scientific computation on GPUs easier to accomplish. NVIDIA released a set of software development tools known as Compute Unified Device Architecture (CUDA) in 2006 that allowed GPU programming on NVIDIA hardware to be implemented via interfacing with the commonly used programming language C. CUDA was later extended to C++.

3.1.1 GPU programming languages

CUDA is not the only language used to program GPUs. Other languages include

- AMD, the main competitor to NVIDIA in graphical hardware, supports GPU programming on AMD hardware via their Stream language.
- OpenCL is a language developed by Apple & other companies that supports GPU programming on different GPU hardware platforms (NVIDIA, AMD). OpenCL code can also be run on CPUs and other specialized processors.

- C++ AMP is a parallel programming library developed by Microsoft to implement parallel programming directly in C++. C++ AMP code can currently only be compiled for Microsoft Windows.

- OpenACC is a standard developed by Cray, NVIDIA and PGI that allows for GPU programming to be done at a very high level using C/C++/Fortran compiler directives.

- Libraries for higher level languages such as R (via the gputools package) and Matlab (via the parallel programming toolbox) have begun to be developed that allow their code to take advantage of GPU programming.

At this time, CUDA is the most well-supported and documented GPU programming language. In addition to base CUDA, NVIDIA and other third party developers provide specialized libraries for specific tasks.

- cuRAND provides functions for random number generation.

- cuBLAS implements basic linear algebra operations as defined by the BLAS standard.

- MAGMA (open source software developed at University of Tennessee), CULA and ArrayFire are third party libraries that provide GPU implementations of matrix operations and decompositions commonly found in LAPACK.

- Thrust is a library of functions and data structures designed to simplify programming in CUDA.

Given its support, CUDA is used for the programming in this paper.

3.2 Programming in CUDA & Thrust

This section will provide a short overview of programming in CUDA, although first a brief discussion must be made of the architecture of a GPU. A GPU is comprised of a number of multiprocessors. The multiprocessor consists of a number of stream processors, also known as cores; the most recent NVIDIA graphics cards, referred to as their Kepler architecture, have 192 cores per multiprocessor. For example, a NVIDIA GTX670 graphics card has 7 multiprocessors and 1,344 cores.

There are also several kinds of memory on a GPU. There is general device memory that is shared by all multiprocessors. Each multiprocessor has its own block of memory, currently at 64 kB, available for use to all of its cores. Furthermore, each core has a number of 32-bit registers, which is memory where arithmetic operations are implemented. To do computation on the GPU, data must first be transferred from the computer's general RAM (known as the host) to the GPU and then transferred to the individual cores. However, the GPU memory must first be reserved.

3.2.1 CUDA programming

In C, to allocate host memory to hold N numbers, it is sufficient to use a line of code similar to:

```
double *data = new double[N];
```

Assuming that this array is filled appropriately, it is copied to GPU memory as follows:

```
double *dataGPU;
cudaMalloc((void**)&dataGPU, N*sizeof(double));
```

```
cudaMemcpy(dataGPU, data, N*sizeof(double), cudaMemcpyHostToDevice);
```

Once data is copied to the GPU, computations are implemented on the GPU via a *kernel* function. For example, assuming that the desired operations are stored in a function `doStuff`, the following code applies this function to every element of the data in parallel:

```
doStuff<<<x, y>>>(data, ...);
```

where ... indicate any other necessary parameters for `doStuff`. Of main interest here are the values x and y; these values are the number of *blocks* and number of *threads*, respectively, to be launched on the GPU.

The kernel function is executed in parallel via threads, where each thread applies the function to a different element of memory; the actual processing of each thread is done on one of the GPU's cores. These threads are grouped into blocks, where all of the threads in one block occurs on one multiprocessor; currently, there can be no more than 1,024 threads per block. Threads within a block can access the memory shared by that block, which generally occurs at very high speeds. Each thread is assigned an ID number that is used to determine to which memory location that thread should apply the kernel function.

It should be noted that the GPU always launches threads in multiples of 32; this is known as a *warp*. When coding in CUDA, it is important to include a test based on the thread ID to make sure that unneeded threads do not write over memory that should be untouched.

Blocks themselves are grouped into grids. Blocks can be run either in parallel or sequentially. The advantage of this is that at runtime the GPU will determine the

number of available multiprocessors, and allocate the blocks in an optimal fashion.

For computers with more advanced graphics cards, the GPU will automatically take advantage of the presence of additional multiprocessors. However, programming directly in CUDA still requires a good understanding of the architecture to maximize performance, primarily by minimizing memory transfers. For example, this can be done via memory padding and breaking large datasets into smaller pieces and repeatedly calling a kernel on the smaller pieces (Suchard et al., 2010). As an alternative, one can code in Thrust, which abstracts and automates a significant portion of memory allocation, instead of coding directly in CUDA.

3.2.2 Thrust programming

In contrast to the previous CUDA code used to transfer memory from host to the GPU, in Thrust this can be implemented in one line:

```
thrust::device_vector<double> dataGPU = data;
```

While the same underlying memory operations are still executed, Thrust wraps these operations in a much simpler framework, making code development easier and faster.

Thrust also provides functions to apply other functions to data stored on the GPU, as opposed to calling the kernel notation `<<< >>>` used in CUDA. The primary two functions used by Thrust for this purpose are `transform` and `for_each`. For example, two arrays, also known as vectors in CUDA/Thrust, of data in GPU memory may be multiplied elementwise and stored in a different array using:

```
thrust::transform(data1.begin(), data1.end(), data2.begin(),
                  result.begin(), thrust::multiplies<double>());
```

The `transform` function allows for at most two input vectors, while the `for_each` function allows only one input vector. Three or more inputs can be used via the use of a `zip_iterator`, which wrap multiple inputs into "one" vector, such that functions like `transform` or `for_each` may be used.

Thrust also provides a number of other useful algorithms, such as sorting vectors, searching vectors for specific elements, summing the elements of a vector, etc. In CUDA, all such operations would need to be coded by the user. This also hints at the ease of using Thrust to process data in parallel as opposed to standard approach of processing data serially on a CPU. One can replace a for loop that applies the same function to every element of an array by loading the data into GPU memory and just applying the appropriate Thrust function.

The downside to this accessibility is a lack of fine control over program execution. The determination of memory transfers discussed in the section on CUDA is handled automatically by the CUDA compiler when using Thrust, although this may not be optimal from an execution standpoint. Also, all functions in Thrust are applied to contiguous blocks of memory stored in vectors; if it is necessary to repeatedly access different pieces of memory in a kernel function, Thrust will not be much assistance. However, the ease of learning and developing in Thrust is worth consideration. The programming for this paper was done in Thrust for this reason.

3.3 Impacts of Code Parallelization

Many researchers have implemented parallel programs to solve problems found in many different disciplines. The following improvements in program running times have been reported as the result of parallelization:

- (Suchard et al., 2010) report speed-up of 100x using CUDA in estimating Gaussian mixture models via a Markov chain Monte Carlo method and comparable improvements for Bayesian expectation maximization.

- (Aldrich, 2013) reports a speed-up from 2x to 2500x using CUDA in estimation of a simple real business cycle model via value function iteration depending on the size of the grid used for the value function iteration.

- (Lee et al., 2010) report a speed-up of 500x for a sequential Monte Carlo sampler for estimating a Gaussian mixture model, and a speed-up of 30x in estimating a factor stochastic volatility model.

- (Hendeby et al., 2007) report a 10x speed-up for a particle filter with 100,000 particles used in a velocity tracking model.

- (Hendeby et al., 2010) report little increase in speed for a particle filter, although they generated the random numbers needed for the particle filter on the host and continuously transferred them to the GPU.

- (Goodrum et al., 2012) report a speed-up of 71x over Matlab and 35x over C using CUDA for a particle filter with 100,000 particles used for object tracking in video.

(Goodrum et al., 2012) also discuss that they programmed their own Gaussian and uniform random number generators for the GPU; random number generation directly on the GPU via CUDA was not supported until recently. They also considered using Thrust, but decided against it; they specifically call out Thrust's inability to access

irregular sections of memory. Given that this paper implements a particle filter in Thrust, it is possible this is related to their random number generation routines.

Other related references include:

- (Murray, 2012) compares a particle filter in Metropolis MCMC sampler implemented on a GPU to a sequential multinomial sampler implemented on a CPU for parameter estimation of a state-space model, and finds the particle filter works faster if the number of particles is less than 4,096. Random numbers for the particle filter were generated on the host and transferred to the GPU.

- (Brun et al., 2002) report a speed-up of 31x for a particle filter using 32,000 particles when going from a 2-processor to 32-processor Cray Origin2000 supercomputer.

3.4 A Parallelized Particle Filter

As mentioned at the end of Chapter 2, the standard SIR particle filter algorithm contains several steps where it is necessary to apply a function to every particle in memory. On a serial CPU, this requires a loop which processes one particle at a time. On a GPU, it is instead possible to operate on multiple particles simultaneously, up to the number of cores available in hardware.

For example, the calculation of particle weights $w_t^i = w_{t-1}^i p(y_t|x_t^i)$ for all particles on a CPU would be implemented using code similar to:

```
for(i = 0; i < N; i++){
   weights[start_t+i] = weights[start_t_minus1 + i] *
```

```
                        probabilities[start_t+i];
};
```

In Thrust, this can be implemented as:

```
thrust::transform(weights.begin()+(t-1)*N, weights.begin()+t*N,
                  density.begin()+t*N,
                  weights.begin()+(t+1)*N,
                  thrust::multiplies<double>());
```

While this code looks somewhat more complex, this is primarily due to the offsets in terms like `t*N`, which are necessary to store the weights for all particles over all time periods. It is not too difficult in practice to go through serial particle filter code and replace `for` loops with appropriate Thrust commands. The following chapters do this for particle filters used in three related scenarios, and investigate the amount of time saved by using Thrust.

4

AR(1) Process with Noise & Known Parameters

4.1 The AR(1) Model with Noise

One of the simplest hidden Markov models is an autoregressive time series model where observed data are measured with errors. This model belongs to the well-studied class of dynamic linear models (Pole et al., 1994). In this model, the true signal in each time period is based on the value in the previous time period plus an additional innovation; the innovation in a given period is assumed to be independent of innovations in all other periods. Observations in each period are equal to the true signal for that period plus a stochastic error due to measurement error. The innovations and errors are commonly assumed to follow a normal distribution with zero mean.

Mathematically, this model can be expressed as:

$$\begin{aligned} x_t &= \phi x_{t-1} + \epsilon_t \\ y_t &= x_t + \nu_t \end{aligned} \tag{4.1}$$

where $\epsilon_t \sim N(0, v)$ and $\nu_t \sim N(0, w)$. It follows from this definition that the distribution of x_t given x_{t-1}, $p(x_t|x_{t-1})$, is $N(\phi x_{t-1}, v)$. Similarly, the distribution of y_t given x_t is $p(y_t|x_t) = N(x_t, w)$. Assuming that x_t is a stationary series, it can be shown that the marginal or unconditional distribution of x_t is $p(x_t) = N(0, s)$ and the marginal distribution of y_t is $p(y_t) = N(0, s + w)$ where $s = v/(1 - \phi^2)$.

This mathematical model has a corresponding graphical model representation displayed in Figure 4.1. It is clear in comparison to the graphical model of hidden Markov models in Figure 2.1 that this model belongs to the class of hidden Markov models. The only difference between these figures is that the additional parameters of the transition and observation distributions are explicitly recognized in the graphical model below. The AR(1) structure of the model is recognizable given that the distribution of each x_t is dependent on only the value of the preceding x_{t-1} for all i.

For now we assume that the parameters ϕ, v, and w are known. This leaves only the values $x_0 : x_t$ as unknowns. Note that:

1. The distribution of x_t depends on a linear transformation of x_{t-1} given by the linear function $f(x_{t-1}) = \phi x_{t-1}$.

2. The distribution of y_t depends on a linear transformation of x_t given trivially by the identity function $g(x_t) = x_t$.

3. By assumption, the transition and observation distributions are both Gaussian.

As a result, inference on the hidden states $x_{1:t}$ may be performed using the Kalman filter, as mentioned in Chapter 2.

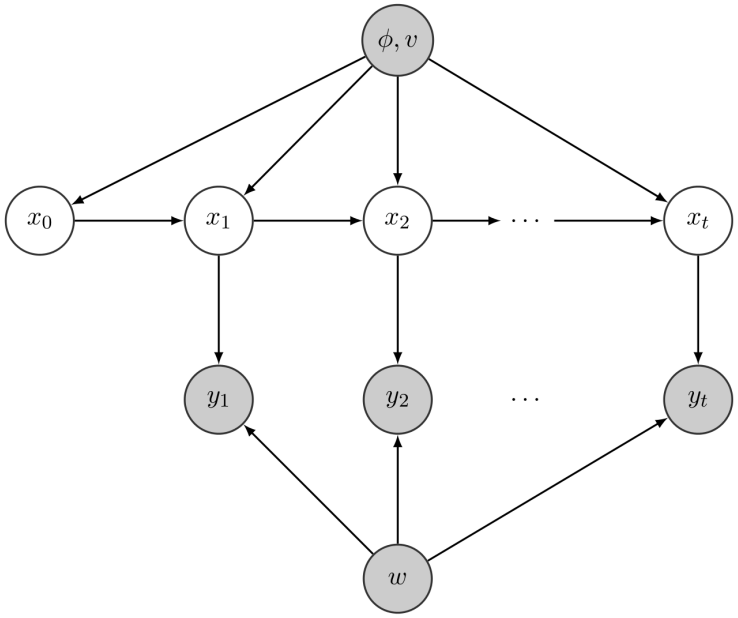


FIGURE 4.1: Graphical model representation of AR(1) process with noise & known parameters

4.2 The Kalman Filter

Details of the Kalman filter can be found in many textbooks (West and Harrison, 1997; Durbin and Koopman, 2012; Shumway and Stoffer, 2010). For reference, the Kalman filter works as follows. Let

$$\begin{aligned} a_t &= E[x_t|y_{1:t}] \\ P_t &= E[(x_t - a_t)^2] \end{aligned} \tag{4.2}$$

The time update equations or prediction phase of the Kalman filter are:

$$\begin{aligned} a_{t|t-1} &= \phi a_{t-1} \\ P_{t_t-1} &= \phi^2 P_{t-1} + v \end{aligned} \tag{4.3}$$

Once these values are calculated for a given t, the measurement update equations or update phase of the Kalman filter are:

$$\begin{aligned} a_t &= \phi a_{t-1} + \frac{\phi^2 P_{t-1} + v}{\phi^2 P_{t-1} + v + w}(y_t - \phi a_{t-1}) \\ P_t &= \phi^2 P_{t-1} + v - \frac{(\phi^2 P_{t-1} + v)^2}{\phi^2 P_{t-1} + v + w} \end{aligned} \tag{4.4}$$

Given selected values for a_0 and P_0, the values of $a_1, a_2, \ldots, a_t$ and $P_1, P_2, \ldots, P_t$ can be calculated iteratively. Given that all error terms are Gaussian distributed, the resulting probability distribution for x_t given $y_{1:t}, \phi, v$ and w is:

$$p(x_t|y_{1:t}, \phi, v, w) = N(a_t, P_t) \tag{4.5}$$

Of interest to this paper, the particle filter can be used to approximate the posterior conditional distribution of hidden states $x_{1:t}$ in lieu of using the Kalman filter. Sequential importance resampling particle filters implemented in R, C++ and Thrust were applied to five hundred randomly simulated time series comprised of twenty periods in order to compare the results and running times of particle filters in the three languages, as well as to compare the results of the particle filter to the Kalman filter.

4.3 Data Simulation

The time series were simulated as follows. Five hundred sets of parameters ϕ, v, w were simulated from the following truncated normal distributions:

$$\begin{aligned} \phi &\sim N_{(0.8,1)}(0.9, 0.05) \\ v &\sim N_{(0,0.1)}(0.05, 0.02) \\ w &\sim N_{(0,0.15)}(0.075, 0.05) \end{aligned} \tag{4.6}$$

Given a set of simulated parameters, a time series of hidden states $x_{1:t}$ is simulated by first simulating x_0 from the marginal $N(0, s)$ distribution of x_t. Subsequent states for $t = 1, 2, \ldots, 20$ are simulated from the conditional distribution $p(x_t|x_{t-1}) = N(\phi x_{t-1}, v)$. The "observed" values $y_{1:20}$ were then simulated from their conditional distribution $p(y_t|x_t) = N(x_t, w)$. This gives a complete time series dataset of both hidden and observed values along with known parameters.

4.4 Comparison of Results

4.4.1 Comparison to the Kalman Filter

The Kalman filter was applied to each time series to determine the marginal conditional mean and variance at each period t in the series. Based on these quantities, the mean absolute error, mean 95% confidence interval width, and coverage ratio are calculated for each time series.

1. The absolute error for each time t is calculated as the absolute value of the difference between the Kalman mean at time t, a_t, and the value of the hidden state x_t. The mean absolute error over all twenty data points is then calculated.

2. The 95% confidence interval width at each time t is calculated as $2 \times 1.96P_t$. The mean interval width over all twenty data points is then calculated.

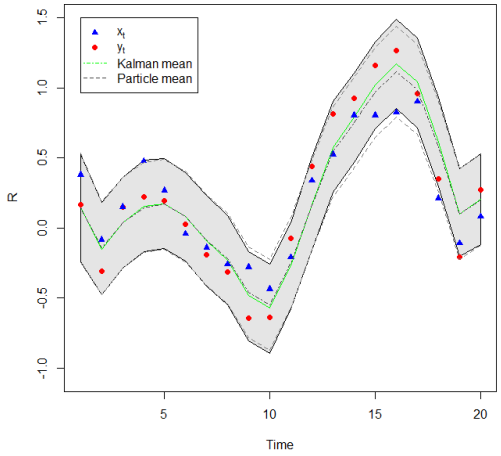
3. The coverage ratio is calculated as the percentage of time periods where the hidden state x_t is within the 95% Kalman confidence interval.

Similarly, the mean absolute error, mean 95% confidence interval width and coverage ratio are calculated for each time series based on the particle filter results.

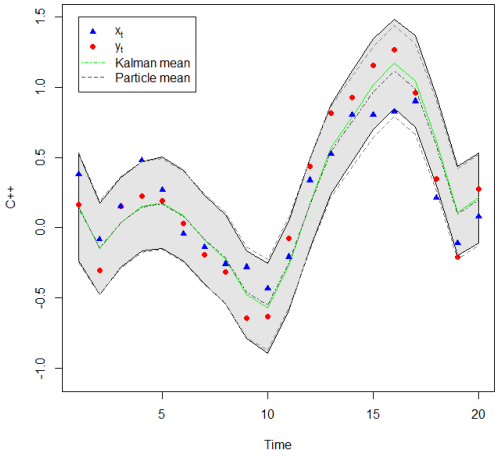
1. The mean particle location at each time period t is calculated as the weighted average of particle locations x_t^i, using particle weights $\tilde{w}_t^i$. The mean absolute error is then calculated as was done for the Kalman filter.

2. At each time period t, the particles x_t^i are sorted and the cumulative weights of all particles is calculated. The particles whose cumulative weights are 0.025 and 0.975 define the 95% particle confidence interval. The mean interval width is then calculated as was done for the Kalman filter.

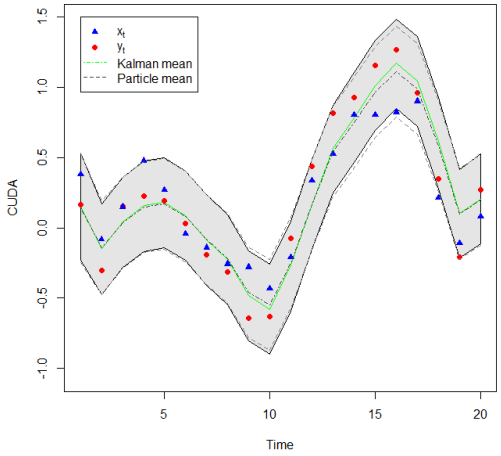
3. The coverage ratio is calculated as the percentage of time periods where the hidden state x_t is within the 95% particle confidence interval.



(a) R

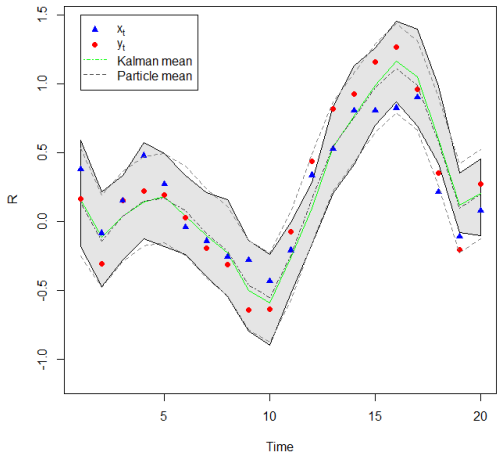


(b) C++

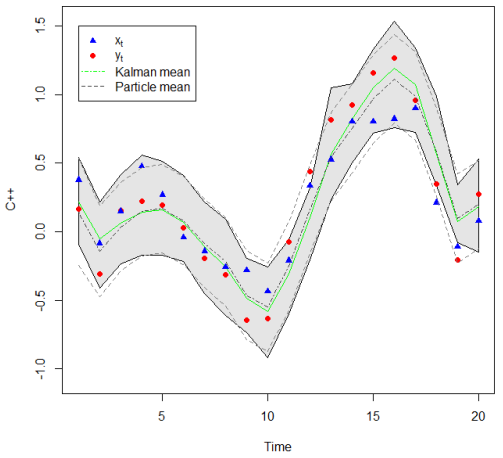


(c) Thrust

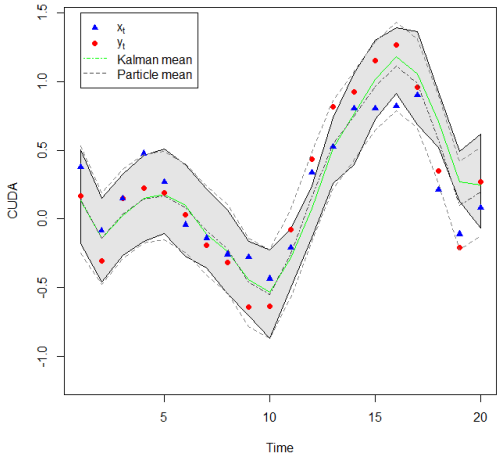
FIGURE 4.2: Comparison of the particle filter using 100,000 particles to the Kalman filter for one simulated time series



(a) R

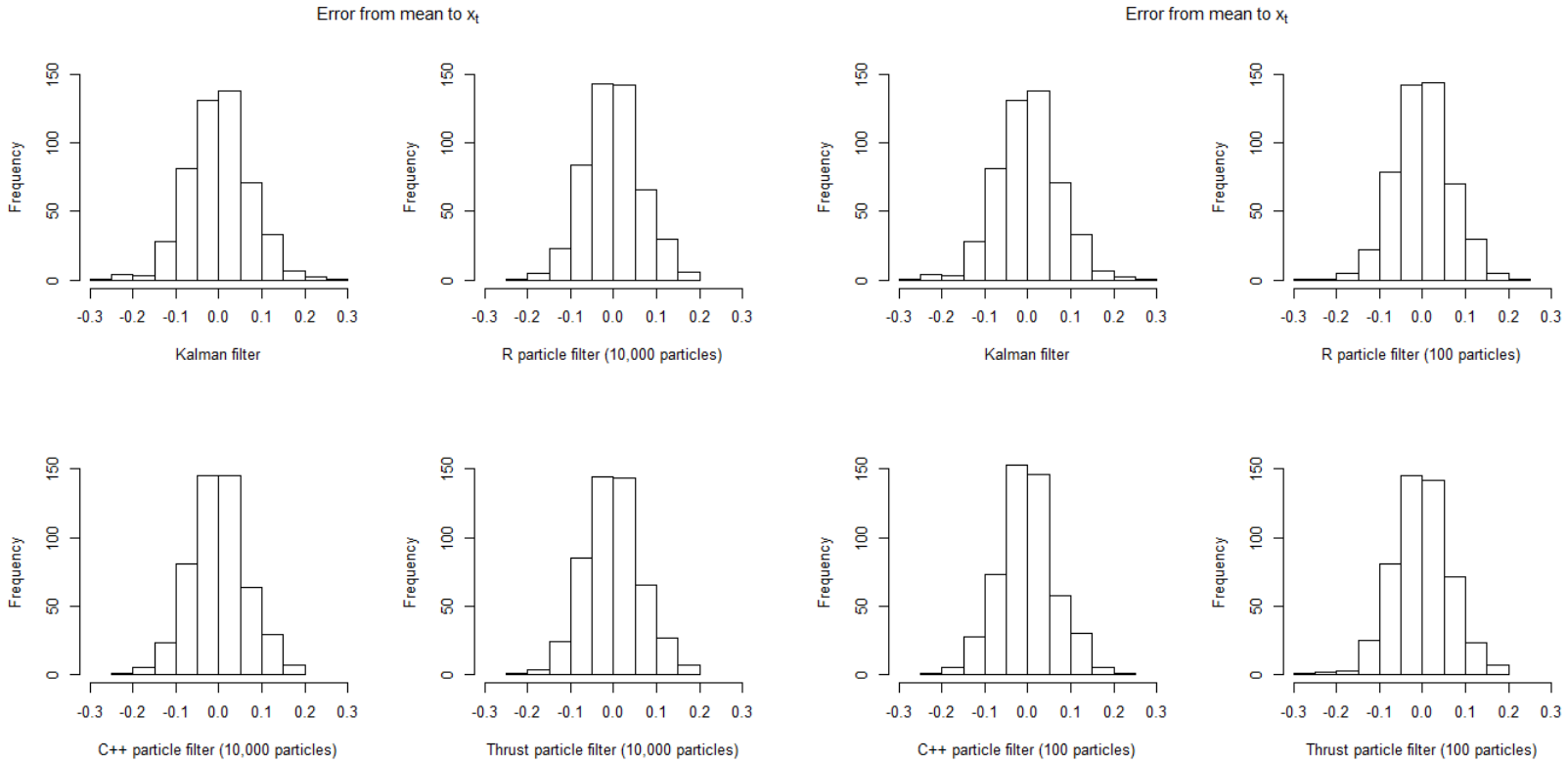


(b) C++



(c) Thrust

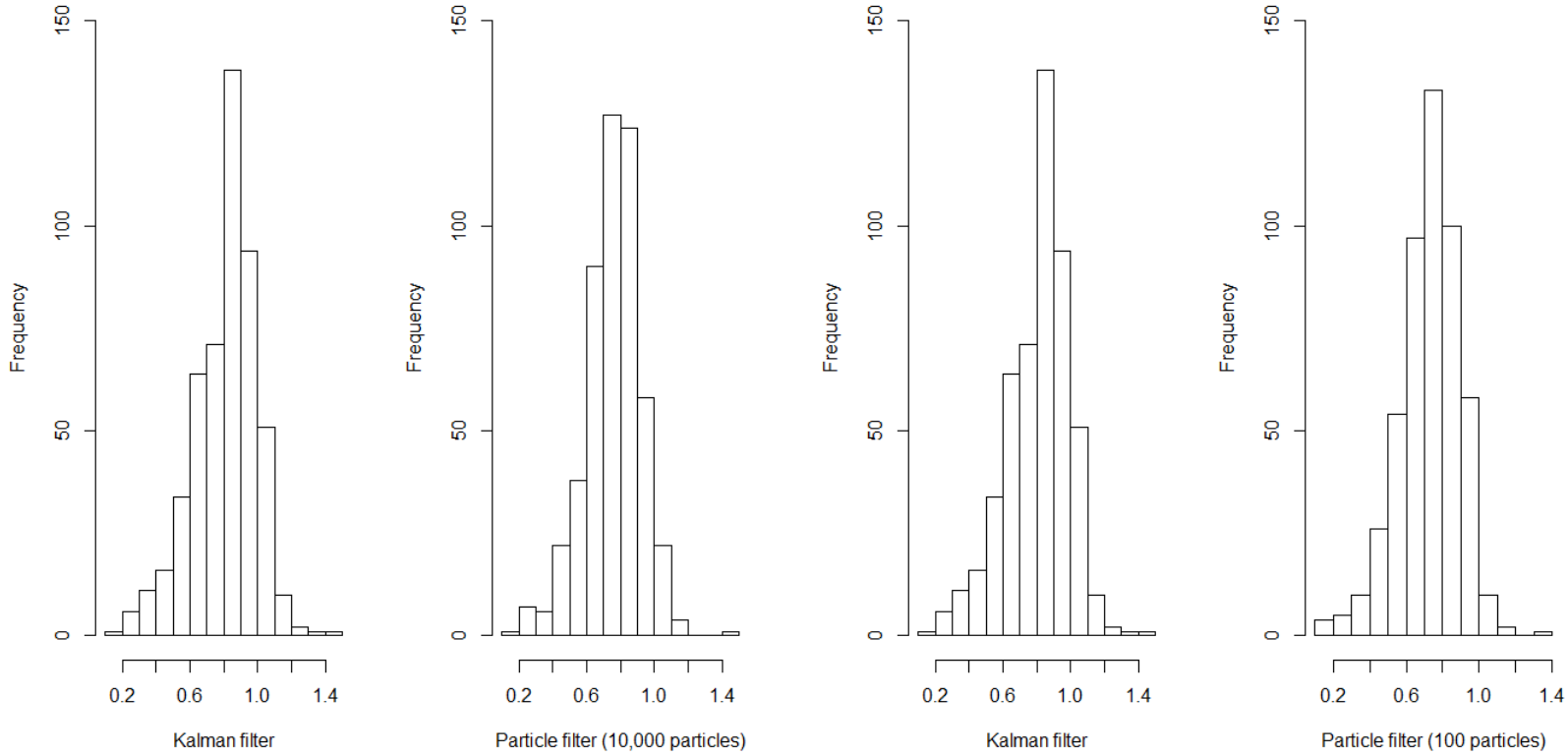
FIGURE 4.3: Comparison of the particle filter using 100 particles to the Kalman filter for one simulated time series



(a) 10,000 particles

(b) 100 particles

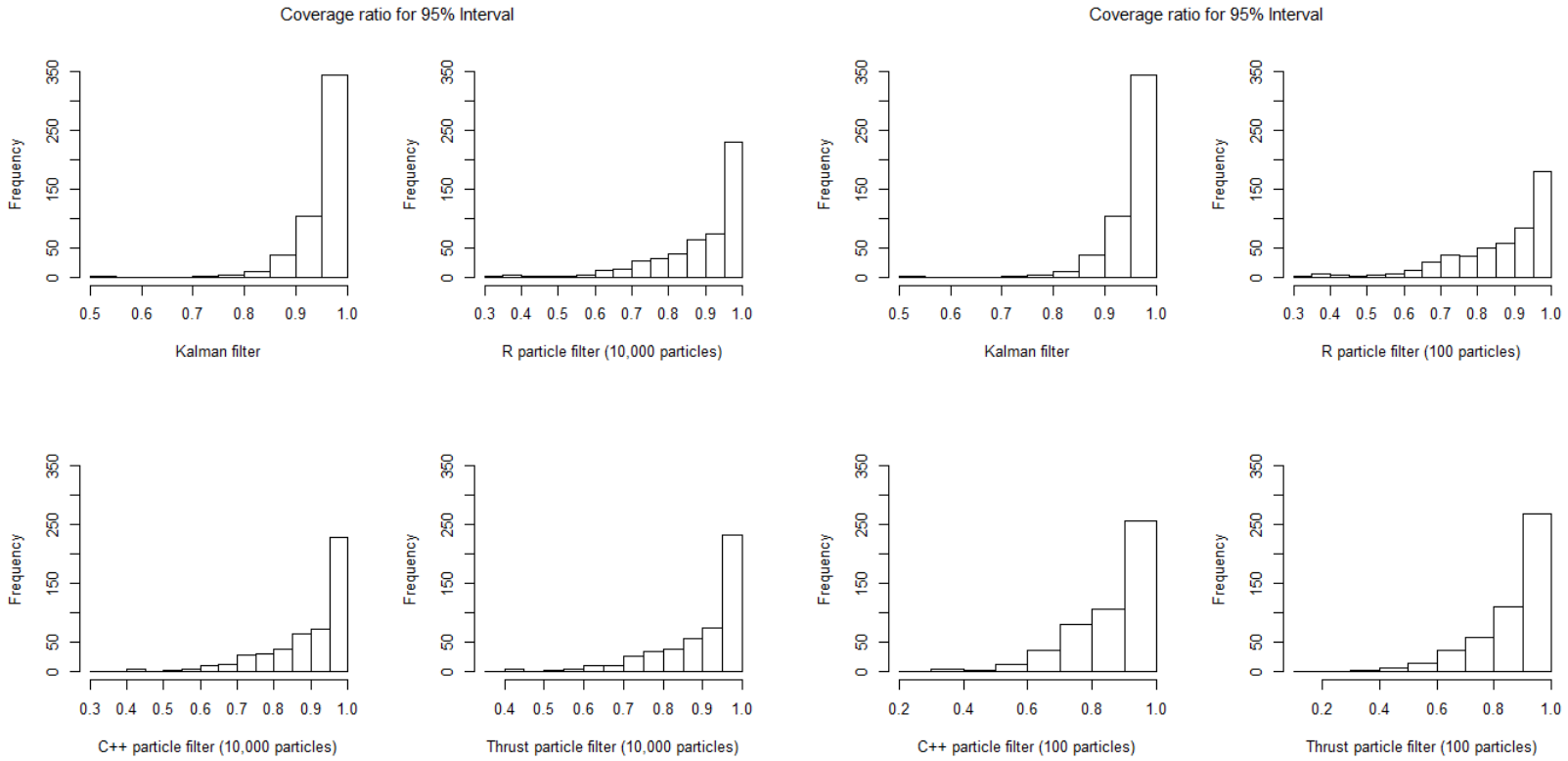
FIGURE 4.4: Average error between the filter mean and the true hidden state x_t



(a) 10,000 particles

(b) 100 particles

FIGURE 4.5: Average interval widths for the Kalman and particle filters



(a) 10,000 particles

(b) 100 particles

FIGURE 4.6: Coverage ratios for the Kalman and particle filters

4.4.2 Timing Comparison

The SIR particle filter was implemented in R, C++ and CUDA/Thrust on a Windows PC with a Intel Core i5-3570K 3.40Ghz processor. The installed graphics card is a NVIDIA Geforce GTX660 Ti GPU with 1,344 cores running at 925Mhz with 2GB of video RAM.

The average running time for a particle filter in the three languages using 100, 1,000, 10,000 and 100,000 particles were:

Table 4.1: Average running time of particle filter in milliseconds

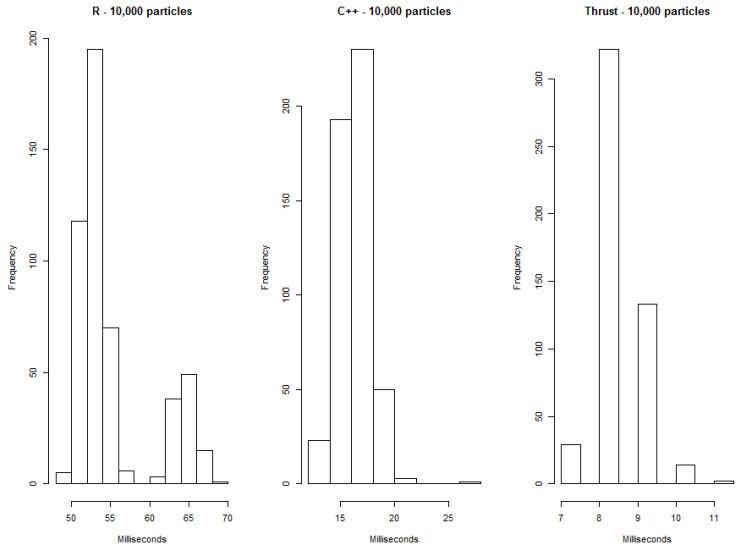
Number of Particles	R		C++		Thrust	
100	1.4		0.1		5.0	
1,000	7.4	(5.3x)	1.4	(14.0x)	6.1	(1.2x)
10,000	55.7	(7.5x)	15.8	(11.3x)	8.3	(1.4x)
100,000	580.2	(10.4x)	201.4	(12.7x)	31.9	(3.8x)

The standard deviation of the running time for a particle filter in the three languages were:

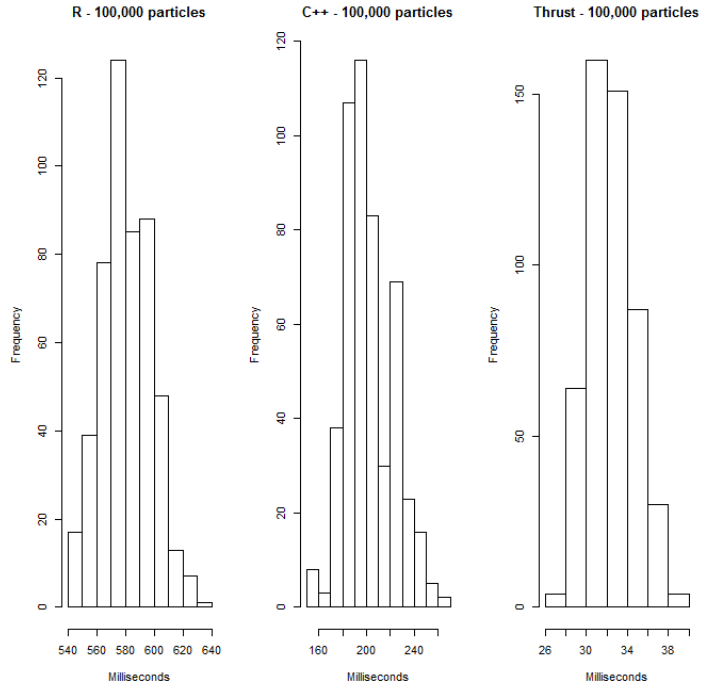
Table 4.2: Standard deviation of running time of particle filter in milliseconds

Number of Particles	R		C++		Thrust	
100	0.8		0.3		0.4	
1,000	3.5	(4.4x)	0.5	(1.7x)	0.5	(1.3x)
10,000	5.0	(1.4x)	1.5	(3.0x)	0.6	(1.2x)
100,000	17.2	(3.4x)	20.1	(13.4x)	2.2	(3.7x)

Histograms displaying running times for 10,000 and 100,000 particles are displayed on the next page.



(a) 10,000 particles



(b) 100,000 particles

FIGURE 4.7: Comparison of the particle filter using 100,000 particles to the Kalman filter for one simulated time series

Timing comparison within a given language

Since several steps in the particle filter that require looping over each particle, the running time of the particle filter should be approximately linear in the number of particles, excluding some fixed overhead for the other steps.

- Going from 100 to 1,000 particles we see a 5.3x, 14.0x, and 1.2x increase in the average running times of R, C++ and Thrust respectively.
- Going from 1,000 to 10,000 particles we see a 7.5x, 11.3x, and 1.4x increase in the average running times of R, C++ and Thrust respectively.
- When going from 10,000 to 100,000 particles we see a 10.4x, 12.7x, and 3.8x increase in the average running times of R, C++ and Thrust respectively.

There is no significant increase in the average runtime of the Thrust implementation until we increase to 100,000 particles. Until that point, the additional cores available on the GPU are able to handle the additional computational load necessary to handle the additional particles. Given that the increase in running time increases as the number of particles increases for Thrust, it is reasonable to assume that increasing the number of particles even further, such as to one million, will cause the increase in average running time to get closer to 10 times the running time of 100,000 particles. As such, the marginal benefit of Thrust is most significant for a number of particles below 100,000.

Timing comparison between languages

When using only 100 particles, the particle filter implemented in Thrust takes longer than either the R or C++ implementation. There is most likely due to the processing overhead in transferring data from the main host memory across the PCI bus to the memory on the graphics card. Increasing the number of particles to 1,000 we see

that the Thrust implementation beats the R implementation on average, although the C++ implementation is still over four times faster than Thrust. However, as the number of particles increases further to 10,000 the Thrust implementation overtakes both R (a speedup of 6.7 times) and C++ (a speedup of 1.9 times). When 100,000 particles are used, Thrust is the clear winner, using an average of 31.9 milliseconds to implement a particle filter, which is a speedup of 18.2 and 6.3 over R and C++ implementations, respectively.

In practice, the difference between 580 milliseconds and 32 milliseconds is obviously not significant in total time saved. The following chapters investigate scenarios where these time savings do have a more noticeable impact.

5

AR(1) Process with Noise and Unknown Parameters

5.1 The AR(1) Model, Revisited

The preceding section assumed that the parameters ϕ, v and w of the AR(1) process were known values. However, in practice one does not know these underlying parameter values when doing inference on the hidden states of the process. As a result, inference must be made on the parameter values as well. There are several approaches that can be used for this situation.

If the hidden states are discrete-valued as opposed to continuous, one can use a maximum-likelihood based approach using an application of the expectation maximization (EM) algorithm (Dempster et al., 1977), known as the Baum-Welch algorithm (Baum et al., 1970). This approach can also be adapted to continuous hidden states (Gharamani and Hinton, 1996).

If the full dataset from time $t = 1$ to T can be used to estimate the parameters

of the model, several Bayesian approaches can be used. This situation is commonly known as offline learning, in contrast to online learning where the information set at time t only consists of data up to t. One approach is variational Bayes EM, which approximates the posterior distribution of parameters and hidden states as a product of marginal distributions over hidden states and parameters (Beal and Ghahramani, 2006).

Another approach is to use a Markov Chain Monte Carlo (MCMC) Gibbs sampler. Here, the hidden states $x_{1:T}$ are sampled from the conditional posterior distribution $p(x_{1:T}|y_{1:T}, \phi, v, w)$. The parameters are then sampled from the conditional posterior distribution $p(\phi, v, w|x_{1:T}, y_{1:T})$. This process is then repeated, alternating draws from each conditional posterior, until a suitable number of draws are made. Jointly, these draws are samples from the joint posterior $p(x_{1:T}, \phi, v, w)|y_{1:T})$, allowing for inference to be made on the parameters (Frühwirth-Schnatter, 2006). A third option is to use an auxiliary particle filter (Pitt and Shephard, 1999) in which the hidden state x_t is extended to include the parameters; these fixed parameters are then assumed to "artificially" evolve over time via some process along with the hidden states. Kernel smoothing over parameter values in particles at time t provides an approximation to the posterior parameter distribution at time t (West, 1993).

The remainder of this paper uses the approach presented in (Fernández-Villaverde and Rubio-Ramírez, 2007), which uses a Metropolis MCMC sampler, where the likelihood of the data given hidden states and parameters is approximated using the output of a particle filter.

5.2 The Metropolis Algorithm

The following is a brief summary of the Metropolis algorithm. Of interest is the posterior distribution $p(\theta|Y)$ for some set of parameters θ given observed data Y. By Bayes theorem,

$$p(\theta|Y) = \frac{p(Y|\theta)p(\theta)}{p(Y)} = \frac{p(Y|\theta)p(\theta)}{\int p(Y|\theta)p(\theta)d\theta}$$

However, the integral in the numerator can be difficult to evaluate, especially as the dimensionality of θ increases. A common solution to this problem is to approximate the posterior distribution $p(Y|\theta)p(\theta)$ by an empirical distribution based on a large set of samples from $p(Y|\theta)p(\theta)$.

One way to construct this set is to start with arbitrary values θ_0. Generate a new sample θ^* in the neighbourhood of θ_0 by drawing from some known, symmetric distribution $q(\theta^*|\theta_0)$. This draw can be accepted as a valid draw from $p(\theta|Y)$ if the probability, conditional on the data Y, of θ^*, $p(\theta^*|Y)$, is close to or greater than the probability of θ_0, $p(\theta_0|Y)$. This is the main idea of the Metropolis algorithm:

The Metropolis algorithm

1. Start with initial parameters θ_0.
2. Let $i = 1$.
3. Sample proposed parameters θ_i^* from $q(\theta^*|\theta_{i-1})$.
4. Calculate an acceptance probability a equal to:

$$a = \frac{p(\theta_i^*|Y)}{p(\theta_{i-1}|Y)} = \frac{p(Y|\theta_i^*)p(\theta_i^*)}{p(Y|\theta_{i-1})p(\theta_{i-1})}$$

5. Let $\theta_i = \theta_i^*$ with probability a. Otherwise, let $\theta_i = \theta_{i-1}$.

6. Repeat steps 3-5 until a large enough set of samples has been collected.

5.3 Particle Filtering Inside a Metropolis Sampler

The distribution $p(\theta)$ is known as the prior distribution for θ, which encapsulates our beliefs about the relative probabilities of different possible values of θ before observing any data. Assuming the prior distribution is vaguely representative of the "true" distribution of the θ, the use of such an informative prior concentrates the posterior distribution in the region of the support of θ that has the highest probability density.

It is possible that the researcher has no prior knowledge about the relative probabilities of different possible values of θ. In this case, the use of a "non-informative prior" for θ where all possible values of θ are considered to be equally likely can be used. In this case, the value of $p(\theta_i^*) = p(\theta_{i-1})$, so the acceptance probability in step 4 of the Metropolis algorithm simplifies to $a = p(Y|\theta_i^*)/p(Y|\theta_{i-1})$. However, $p(Y|\theta)$ is exactly the likelihood function that is key to both frequentist and Bayesian statistical inference.

The output of the particle filter in a hidden Markov model is a set of particles $x_t^{(i)}$ and weights $w_t^{(i)}$ for $i = 1, \ldots, N$ at each period t that, conditional on parameters and observed data up to period t, approximate the distribution of the hidden state x_t. This output can be summarized as $S = \left\{ \{w_t^i, x_t^i\}_{i=1}^N \right\}_{t=1}^T$.

Given this output, $p(Y|\theta)$ can be approximated using equation 5.1 (Fernández-

Villaverde and Rubio-Ramírez, 2007).

$$\hat{p}(Y|\theta) \approx \prod_{t=1}^{T} \frac{1}{N} \sum_{i=1}^{N} p(Y|w_t^i, x_t^i, \theta) \tag{5.1}$$

This approximation is then incorporated into the Metropolis sampler as a substitute for the true likelihood function:

The Metropolis algorithm with particle filter approximation

1. Start with initial parameters θ_0.
2. Run a particle filter, assuming the parameters θ_0 to be known, to determine the set S of particles for $t = 1, \ldots, T$.
3. Using S, calculate the likelihood of observed data Y, $\hat{p}(Y|\theta_0)$ using equation 5.1.
4. Let $i = 1$.
5. Sample proposed parameters θ_i^* from $q(\theta^*|\theta_{i-1})$.
6. Run a second particle filter, now assuming the parameters θ_i^* are known, to determine a second set S' of particles for for $t = 1, \ldots, T$.
7. Using S', calculate the likelihood of observed data Y, $\hat{p}(Y|\theta_i^*)$
8. Calculate an acceptance probability a equal to:

$$a = \frac{\hat{p}(Y|\theta_i^*)}{\hat{p}(Y|\theta_{i-1})}$$

9. Let $\theta_i = \theta_i^*$ with probability a. Otherwise, let $\theta_i = \theta_{i-1}$.
10. Repeat steps 5-9 until a large enough set of samples has been collected.

A MCMC sampler is typically run for thousands of iterations, where each iteration requires a particle filter to be run. It is for this reason that the potential improvement in running times for particle filters using parallel computation become important. Improvements on the order of seconds for a single particle filter implemented in parallel become much more significant when summed over thousands of times.

5.4 Running Times for the Metropolis algorithm with particle filter approximation

To quantify these improvements, this algorithm was again implemented in R, C++ and Thrust. Each implementation was run for 10, 100 and 500 MCMC iterations. For each number of iterations, particle filters using 100, 1,000, 10,000 and 50,000 particles were used to approximate the likelihood function. Each combination of language, iterations and particles was run ten times, and the average running times were calculated.

The results in seconds in R are:

Table 5.1: R timing for AR(1) simulated data with noise

	MCMC Iterations		
Number of Particles	10	100	500
100	0.372	0.456	0.873
1,000	0.406	0.752	2.391
10,000	0.775	3.311	16.443
50,000	2.886	22.735	116.931

The results in C++ are:

Table 5.2: C++ timing for AR(1) simulated data with noise

	MCMC Iterations		
Number of Particles	10	100	500
100	0.026	0.195	0.813
1,000	0.048	0.356	1.606
10,000	0.277	2.551	12.832
50,000	1.390	14.524	69.104

The results in Thrust are:

Table 5.3: Thrust timing for AR(1) simulated data with noise

	MCMC Iterations		
Number of Particles	10	100	500
100	0.096	0.827	4.098
1,000	0.105	0.990	4.889
10,000	0.139	1.323	6.424
50,000	0.293	2.763	13.555

Timing comparison within a given language

Each MCMC iteration is by its nature a serial process, where the same code gets executed in each iteration. Therefore, we would expect that running times would be linear in the number of iterations, ignoring fixed cost overhead in code outside the MCMC loop. Table 5.4 shows the increase in running times when going from one number of iterations to the next higher number of iterations, as a factor of the running time at the lower number of iterations.

Table 5.4: Increases in running times for Metropolis sampler as a function of the increase in iterations

	Increase in the number of iterations					
	R		C++		Thrust	
Number of Particles	10→100	100→500	10→100	100→500	10→100	100→500
100	1.23	1.91	7.44	4.17	8.61	4.96
1,000	1.85	3.18	7.42	4.51	9.45	4.94
10,000	4.27	4.97	9.20	5.03	9.50	4.86
50,000	7.88	5.14	10.45	4.76	9.43	4.91

For C++ and for Thrust we see this approximately holds true, especially at higher number of particles where more time in code execution is spent inside the Metropolis sampler. For R, the linear relationship does not follow as closely, although R code execution generally involves large amounts of overhead.

For the R and C++ code which is implemented in serial, we expect code running times to also increase approximately linear with the number of particles, because the majority of work done in the Metropolis sampler is committed to looping over the number of particles. Thrust should show again much lower increase in running times as the additional cores of the GPU are utilized.

Table 5.5: Increases in running times for Metropolis sampler as a function of the increase in particles

	R			C++			Thrust		
Increase in Particles	10	100	500	10	100	500	10	100	500
100→1,000	1.09	1.65	2.74	1.83	1.83	1.97	1.09	1.20	1.19
1,000→10,000	1.91	4.40	6.88	5.78	7.16	7.99	1.33	1.34	1.31
10,000→50,000	3.72	6.87	7.11	5.02	5.69	5.39	2.10	2.09	2.11

None of the implementations show a linear increase in running times when going from 100 to 1,000 particles, indicating that the overhead involved in running the

Metropolis sampler is more computationally intensive than the loops over particles for low numbers of particles. As the particles increase, we do see that running times increase approximately linearly for R and C++, while Thrust shows sub-linear increases as expected.

Table 5.6: Comparison of running times for Metropolis sampler across languages, using C++ as a base

	Number of iterations					
	R			Thrust		
Number of Particles	10	100	500	10	100	500
100	14.19	2.34	1.07	3.66	4.24	5.04
1,000	8.46	2.11	1.49	2.18	2.78	3.04
10,000	2.80	1.30	1.28	0.50	0.52	0.50
50,000	2.08	1.57	1.69	0.21	0.19	0.20

Table 5.6 shows the ratio of the average running times for R and Thrust, relative to the average running time for C++, for each combination of number of iterations and number of particles. We again see that at low number of particles that Thrust performs worse than C++, but up to five times better than C++ as the number of particles increases.

6

A Non-Gaussian AR(1) Model

6.1 The Modified Normal-Laplace Distribution

The AR(1) model can be easily extended to be non-Gaussian by simply adding additional innovation terms, as suggested by Juan Rubio-Ramirez:

$$y_t = \rho_y y_{t-1} + (1-\rho_y)\upsilon + (1-\rho_y^2)^{1/2}\exp(\tau)U_t + \exp(\alpha)E_{1,t} - \exp(\beta)E_{2,t} \qquad (6.1)$$

Here, U_t is a standard Gaussian random variable and $E_{1,t}$ and $E_{2,t}$ are exponential random variables with unit rate parameters, with $U_t, E_{1,t}$, and $E_{2,t}$ all mutually independent. This distribution is similar to the Normal-Laplace distribution (Reed and Jorgensen, 2004), and will be henceforth referred to as the modified Normal-Laplace distribution (mNL).

The conditional mean and variance of y_t given y_{t-1} and parameters are:

$$E[y_t|y_{t-1},\rho,\upsilon,\tau,\alpha,\beta] = \rho y_{t-1} + (1-\rho)\upsilon + \exp(\alpha) - \exp(\beta)$$

$$Var[y_t|y_{t-1},\rho,\upsilon,\tau,\alpha,\beta] = (1-\rho^2)\exp(2\tau) + \exp(2\alpha) - \exp(2\beta)$$

The additional terms $\upsilon, \tau, \alpha, \beta$ provide the model additional flexibility in the following ways:

- The υ term incorporates a long-term mean into the process of y_t. If ρ_y was zero, indicating that y_t was conditionally independent of y_{t-1} given knowledge of other parameters, the value of y_t would υ plus random innovations each period.

- The τ term controls the variance of the normally distributed innovations each period.

- The α and β terms contribute positive and negative skewness, respectively, to the innovations to y_t observed each period. If the value of α is larger than the value of β, the innovations are expected to be positive; conversely, the expected innovations will be negative if β is larger than α.

- Also, as α and β increase to positive ∞, the distribution of innovations will become more peaked at zero, thereby increasing the kurtosis of the distribution.

Assuming $y_{t-1} = 1$, three examples of the distribution of y_t for varying values of $\rho, \upsilon, \tau, \alpha, \beta$ can be seen in Figure 6.1. The solid line corresponds to an mNL distribution with the displayed parameters, while the dotted line corresponds to a Gaussian distribution with the same mean and variance as the mNL distribution.

This model as currently constructed includes no hidden states, and therefore inference on parameters can be performed in a relatively straight-forward manner using either maximum likelihood or Bayesian inference. However, the model can be made even more flexible by allowing the parameters υ, τ, α and β to vary over time.

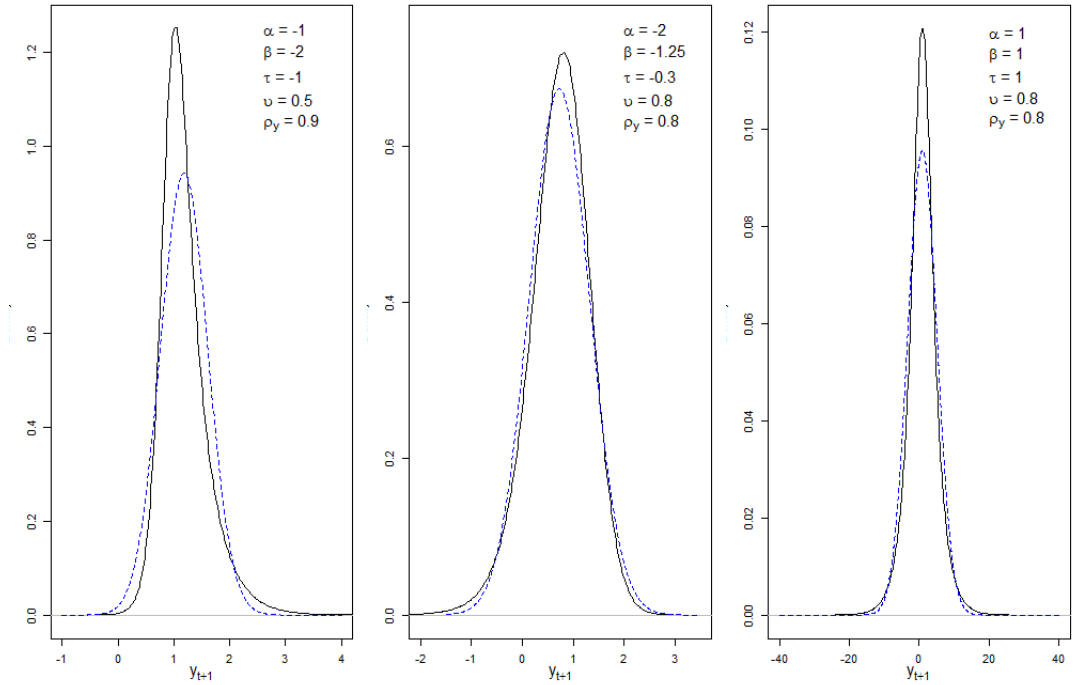


FIGURE 6.1: Examples of a non-Gaussian distribution

In this case, the model becomes:

$$y_t = \rho_y y_{t-1} + (1-\rho_y)\upsilon_t + (1-\rho_y^2)^{1/2}\exp(\tau_t)U_{y,t} + \exp(\alpha_t)E_{1,t} - \exp(\beta_t)E_{2,t} \quad (6.2)$$

These time-varying parameters are now hidden states, since their values are not observed, but impact the distribution of the observed values $y_t, t = 1, \ldots, T$.

To complete the model specification, it is necessary to specify a model for the transition densities for the hidden states. We assume that the hidden states υ, τ, α

and β obey the following laws of motion:

$$\begin{aligned}
\upsilon_t &= (1-\rho_\upsilon)\bar{\upsilon} + \rho_\upsilon \upsilon_{t-1} + \eta_\upsilon U_{\upsilon,t} \\
\tau_t &= (1-\rho_\tau)\bar{\tau} + \rho_\tau \tau_{t-1} + \eta_\tau U_{\tau,t} \\
\alpha_t &= (1-\rho_\alpha)\bar{\alpha} + \rho_\alpha \alpha_{t-1} + \eta_\alpha U_{\alpha,t} \\
\beta_t &= (1-\rho_\beta)\bar{\beta} + \rho_\beta \beta_{t-1} + \eta_\beta U_{\beta,t}
\end{aligned} \tag{6.3}$$

where $\Phi = \{\bar{\upsilon}, \rho_\upsilon, \eta_\upsilon, \bar{\tau}, \rho_\tau, \eta_\tau, \rho_y, \bar{\alpha}, \rho_\alpha, \eta_\alpha, \bar{\beta}, \rho_\beta, \eta_\beta\}$ is the set of 13 unknown parameters in the model. These formulas are equivalent to stating that the transition densities are:

$$\begin{aligned}
p(\upsilon_t|\upsilon_{t-1}, \Phi) &\sim N((1-\rho_\upsilon)\bar{\upsilon} + \rho_\upsilon \upsilon_{t-1}, \eta_\upsilon^2) \\
p(\tau_t|\tau_{t-1}, \Phi) &\sim N((1-\rho_\tau)\bar{\tau} + \rho_\tau \tau_{t-1}, \eta_\tau^2) \\
p(\alpha_t|\alpha_{t-1}, \Phi) &\sim N((1-\rho_\alpha)\bar{\alpha} + \rho_\alpha \alpha_{t-1}, \eta_\alpha^2) \\
p(\beta_t|\beta_{t-1}, \Phi) &\sim N((1-\rho_\beta)\bar{\beta} + \rho_\beta \beta_{t-1}, \eta_\beta^2)
\end{aligned} \tag{6.4}$$

This model can be represented with the following graphical model, where $\mathbf{X_t}$ corresponds to the set of hidden states $\{\upsilon_t, \tau_t, \alpha_t, \beta_t\}$ is displayed in Figure 6.2.

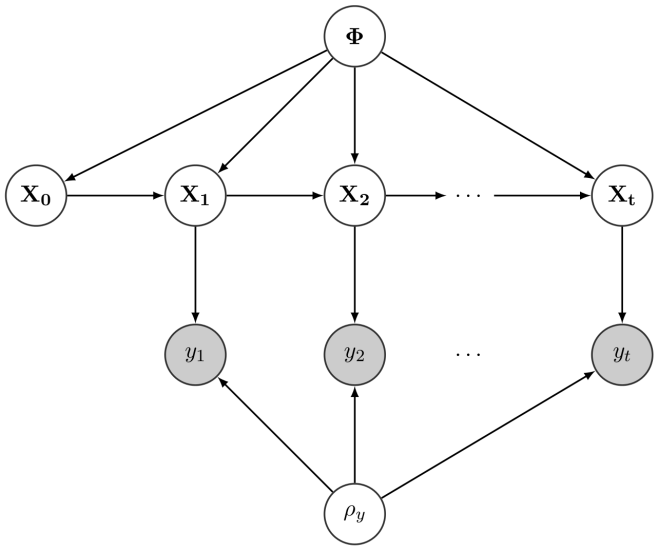


FIGURE 6.2: Simplified graphical model representation of a modified Normal Laplace process with time varying parameters

We can see that the model fits into the same structure as the AR(1) model with noise presented in Chapters 4 and 5. This graphical model structure is a simplification to the true structure, since each node $\mathbf{X_i}$ actually consists of four separate nodes corresponding to the four hidden states at each time period, each connected to the corresponding hidden states at time $i-1$ and $i+1$. The node Φ consists of the 12 unknown parameters, excluding ρ_y, separated into 4 groups of three parameters, each of which is connected to each corresponding hidden state for all time periods. A more accurate representation of this model is shown in Figure 6.3.

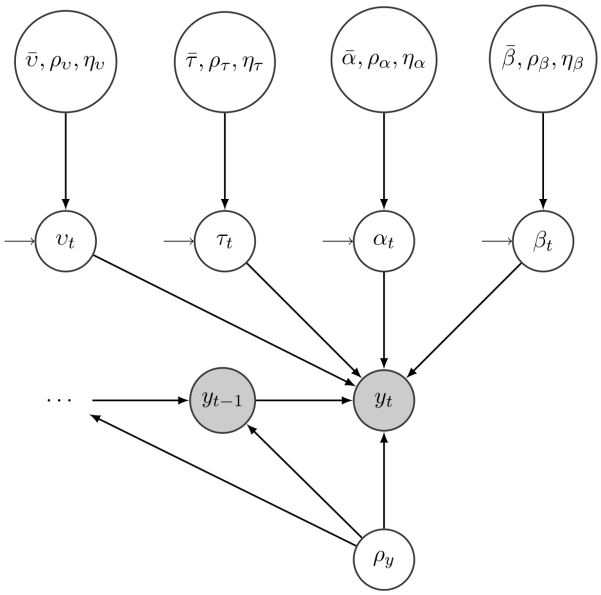


FIGURE 6.3: Graphical model representation of a modified Normal Laplace process with time varying parameters

To perform inference on the parameters of this model, we apply the particle filter in Metropolis-Hastings Markov Chain Monte Carlo approach as presented in Chapter 5. The only modifications necessary are that a particle consists of a four-dimensional vector, corresponding to the four hidden states, and there are thirteen parameters in the model instead of three.

6.2 Comparison of Results

6.2.1 Data Simulation

The data used for the comparison was one simulated time series consisting of 500 periods from using the following set of parameters: In the AR(1) model with noise, we began our simulations by simulating the first hidden state x_0 from the uncon-

Table 6.1: Parameters used to simulate a modified Normal Laplace time series

$\bar{\upsilon} = 0.05$	$\bar{\tau}$= -0.20	$\bar{\alpha}$ = -1.75	$\bar{\beta}$ = -2.00
$\rho_{\upsilon} = 0.92$	$\rho_{\tau} = 0.70$	$\rho_{\alpha} = 0.95$	$\rho_{\beta} = 0.88$
$\eta_{\upsilon} = 0.10$	$\eta_{\tau} = 0.05$	η_{α}= 0.10	$\eta_{\beta} = 0.05$
$\rho_y = 0.94$			

ditional marginal distribution $p(x_t)$. For the modified Normal Laplace distribution, the joint conditional distribution equals the product of the individual conditional distributions:

$$p(\upsilon_t, \tau_t, \alpha_t, \beta_t | \upsilon_{t-1}, \tau_{t-1}, \alpha_{t-1}, \beta_{t-1}) = p(\upsilon_t|\upsilon_{t-1})p(\tau_t|\tau_{t-1})p(\alpha_t|\alpha_{t-1})p(\beta_t|\beta_{t-1})$$

However, it is not simple to derive the unconditional marginal distribution of $p(\upsilon_t, \tau_t, \alpha_t, \beta_t)$ for this model. Therefore, instead of drawing directly from the marginal distribution, we start by setting $\upsilon_0, \tau_0, \alpha_0$ and β_0 to 0; we then iteratively make a large number of draws from the conditional distributions $p(x_t^0|x_{t-1}^0)$, where the superscript indicates that these draws are all considered to be part of the initial state. When finished, the sequence of states prior to the last state drawn is discarded, so the last state drawn can be considered as a draw from the marginal distribution of the hidden states.

Given this "draw" from the marginal distribution of hidden states, subsequent hidden states for periods $t = 1, \ldots, 1,000$ are drawn from their respective conditional distributions. At each time period, given the current hidden states, observed data y_t is drawn from the modified Laplace Normal distribution given by equation 6.2. The result of this procedure is shown in Figure 6.4.

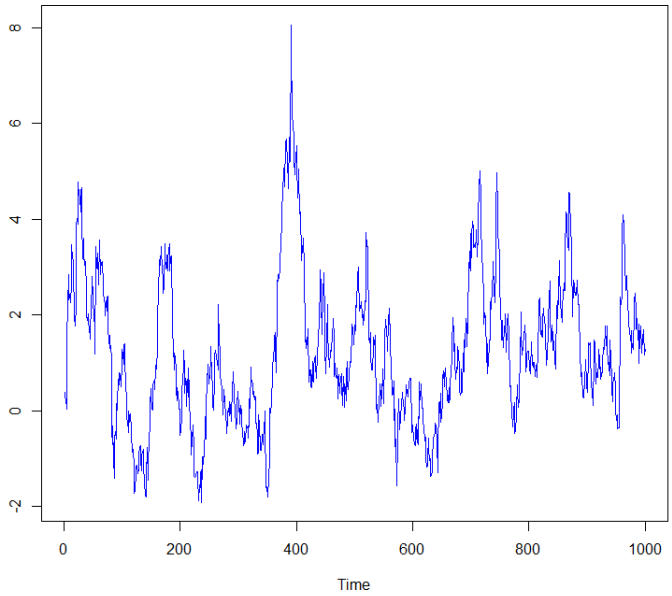


FIGURE 6.4: Simulated time series from a modified Laplace Normal AR(1) time series

6.2.2 Timing Comparison

The results in R are, in seconds:

Table 6.2: R timing for mLN simulated data

	Number of Particles		
MCMC Iterations	100	1,000	10,000
10	39.1	360.6	3,537.6
100	361.7	3,684.6	36,943.8
250	886.8	9,098.4	89,801.4*

These results are based on the average of 10 runs, except for the result for 250 iterations with 10,000 particles. That result is based on only a single run.

The results in C++ are:

Table 6.3: C++ timing for mLN simulated data

	Number of Particles		
MCMC Iterations	100	1,000	10,000
10	0.9	9.1	93.6
100	8.2	84.5	844.0
250	20.6	207.3	2,149.8

The results in Thrust are:

Table 6.4: Thrust timing for mLN simulated data

	Number of Particles		
MCMC Iterations	100	1,000	10,000
10	2.5	3.4	8.9
100	24.7	31.8	80.1
250	58.2	78.5	199.6

In comparing R to C++, we see that the implementation in C++ is on average 42 times faster than in R. This result is fairly consistent across number of iterations and number of particles used. The slow nature of loops in R lead it to be much slower than C++; in C++, memory operations can be avoided by passing memory addresses or references to data instead of copying data, as is done in R.

At small number of particles, the C++ implementation is also three times as fast as the parallel Thrust implementation, once again most likely due to the startup costs associated with running programs on a GPU. However, once 1,000 particles are used, the Thrust implementation becomes 2.7 times faster than C++. This performance gap widens to be 10.6 times once 10,000 particles are used, as the parallel nature of the GPU begins to be fully utilized.

To investigate this further, 10 MCMC iterations were also run in C++ and Thrust using 50,000 particles. The average C++ running time was 485.67 seconds, whereas Thrust took 33.25 seconds on average. The first thing to note is that the C++ running time is approximately five times as long as it was for 10,000 particles; at this stage, the running time is dominated by loops over particles, so any increase in the number of particles should lead to a linear increase in running time. Likewise, Thrust takes approximately 3.74 times as long with 50,000 particles as it does with 10,000 particles; at this number of particles, we begin to reach hardware limitations as the number of cores on the GPU become fully utilized. Therefore, increasing the number of particles beyond this stage should also lead to linear increases in running times.

Also, the results also show that increases in the number of MCMC iterations lead to linear increases in running times, as expected. Therefore, we expect that running 20,000 MCMC iterations for 10,000 particles would take 83 days in R, 2 days in C++ and only 4 hours in Thrust.

As an aside, the effect of using single-precision versus double-precision formats on running times was also looked at. The above times were all run using 64-bit double-precision numbers. The test cases of 10 MCMC iterations with 50,000 particles were rerun using single-precision numbers in C++ and Thrust. The C++ implementation took 354.21 seconds on average, for a speed-up of 37% over the double-precision average of 485.67 seconds. The Thrust implementation took 12.87 seconds, for a speed-up of 158%. This result will be hardware dependent, since the GPU used for testing has hardware limited double-precision throughput; more expensive NVIDIA GPUs have better double-precision performance.

6.2.3 Markov Chain Monte Carlo Diagnostics

To evaluate the performance of the particle filter in MCMC for the modified Normal Laplace AR(1) model, the particle filter in MCMC method was run for 20,000 iterations using 50,000 particles. The results are shown in Figure 6.5. The red vertical lines indicate the true parameters used to simulate the data. It is important to note that the histograms are an approximation to the true marginal posterior distributions of the parameters of the model given the observed data y; the objective of the MCMC algorithm is not to recover the true parameters. As such, it is not necessarily an error that the true values do not occur near a mode of the marginal posteriors.

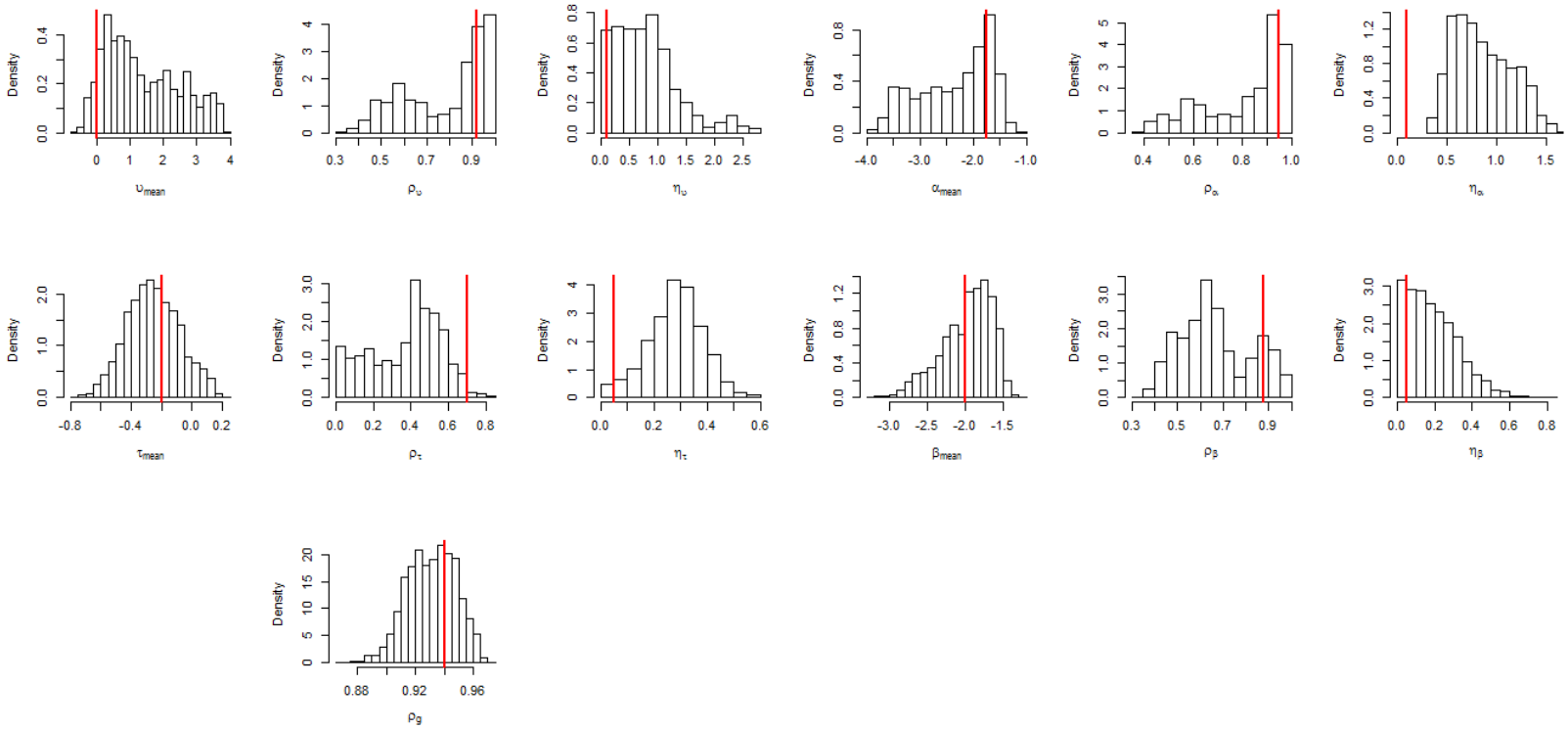


FIGURE 6.5: Bayesian histograms of the mLN parameter estimates for simulated data

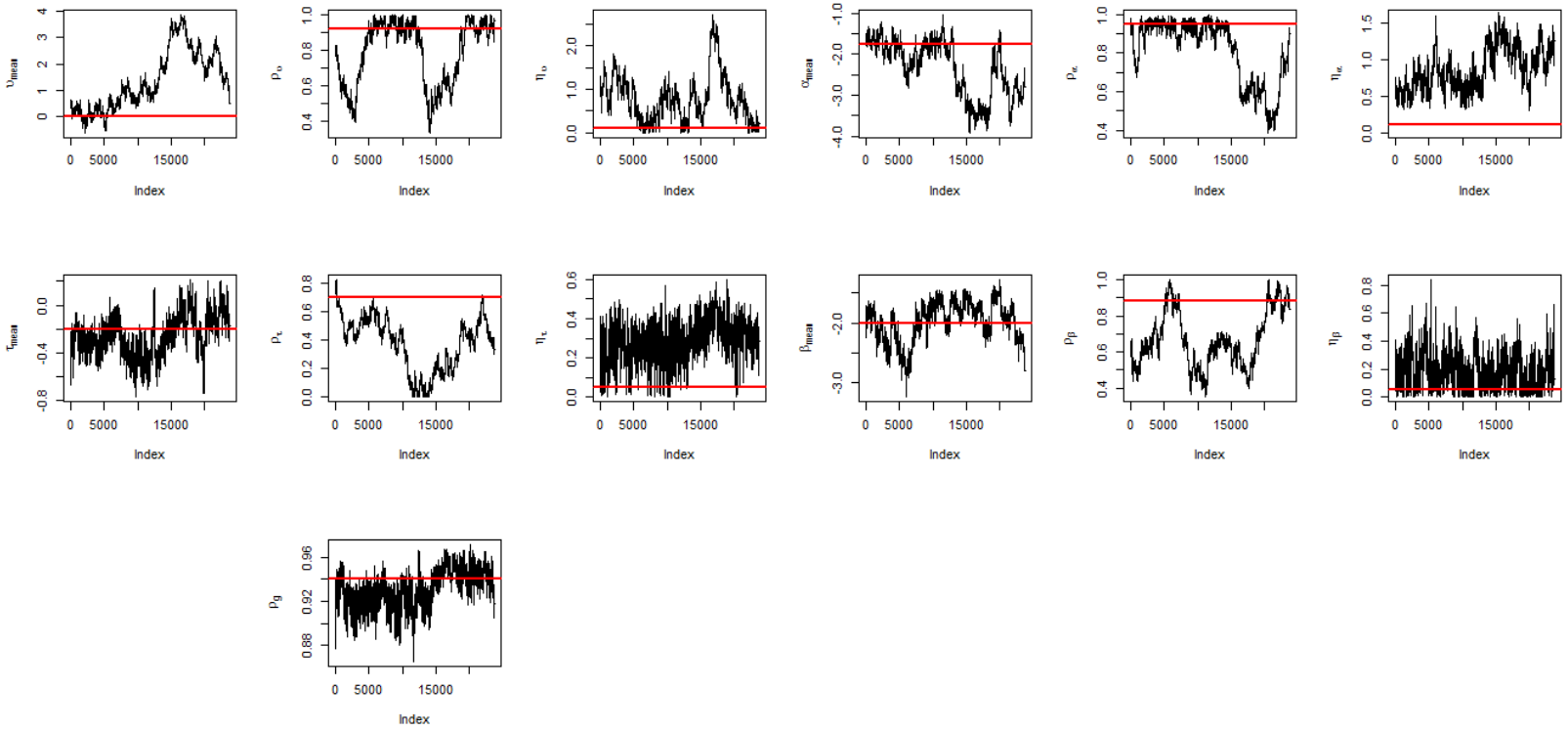
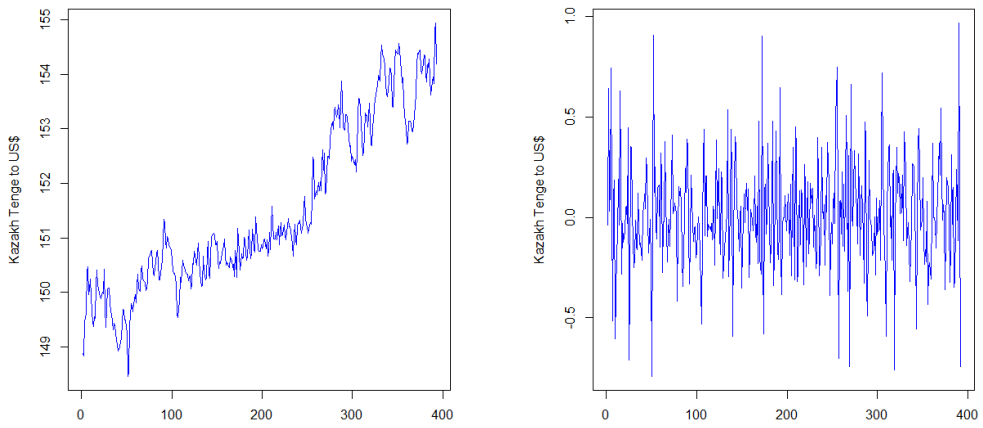


FIGURE 6.6: Trace plots of the mLN parameter estimates for simulated data

6.3 An Application to Currency Exchange Rates

We conclude this section by applying the modified Laplace Normal AR(1) model to the exchange rate of the Kazakhstan tenge to the United State dollar between July 1st, 2012 and January 1st, 2014. The exchange rate between these time periods is shown in Figure 6.7a. This time series is clearly non-stationary over this period, so the series is differenced once, which results in the series displayed in 6.7b.



(a) (b)

FIGURE 6.7: Exchange rate (a) and differenced exchange rate (b) between the Kazakhstan tenge and the U.S. dollar

The differenced time series does appear to be stationary, with some intermittent periods of high volatility. It is not from the plot itself whether the results are positively or negatively skewed, so we estimate the parameters underlying a modified Laplace Normal process for this data. One hundred thousand MCMC iterations were run using 50,000 particles, with the first 5,000 iterations discarded as a burn-in period.

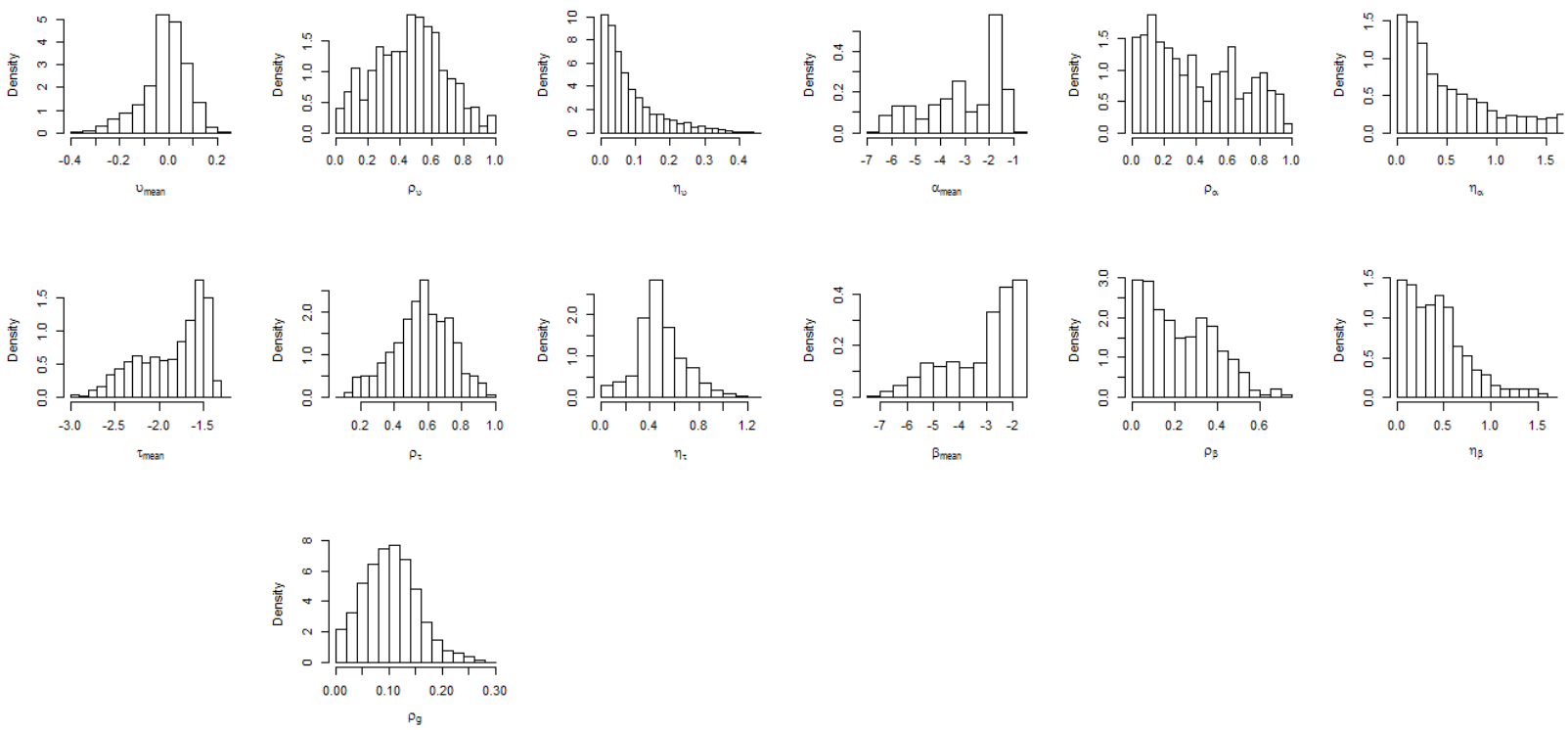


FIGURE 6.8: Bayesian histograms of the mLN parameter estimates for exchange rate data

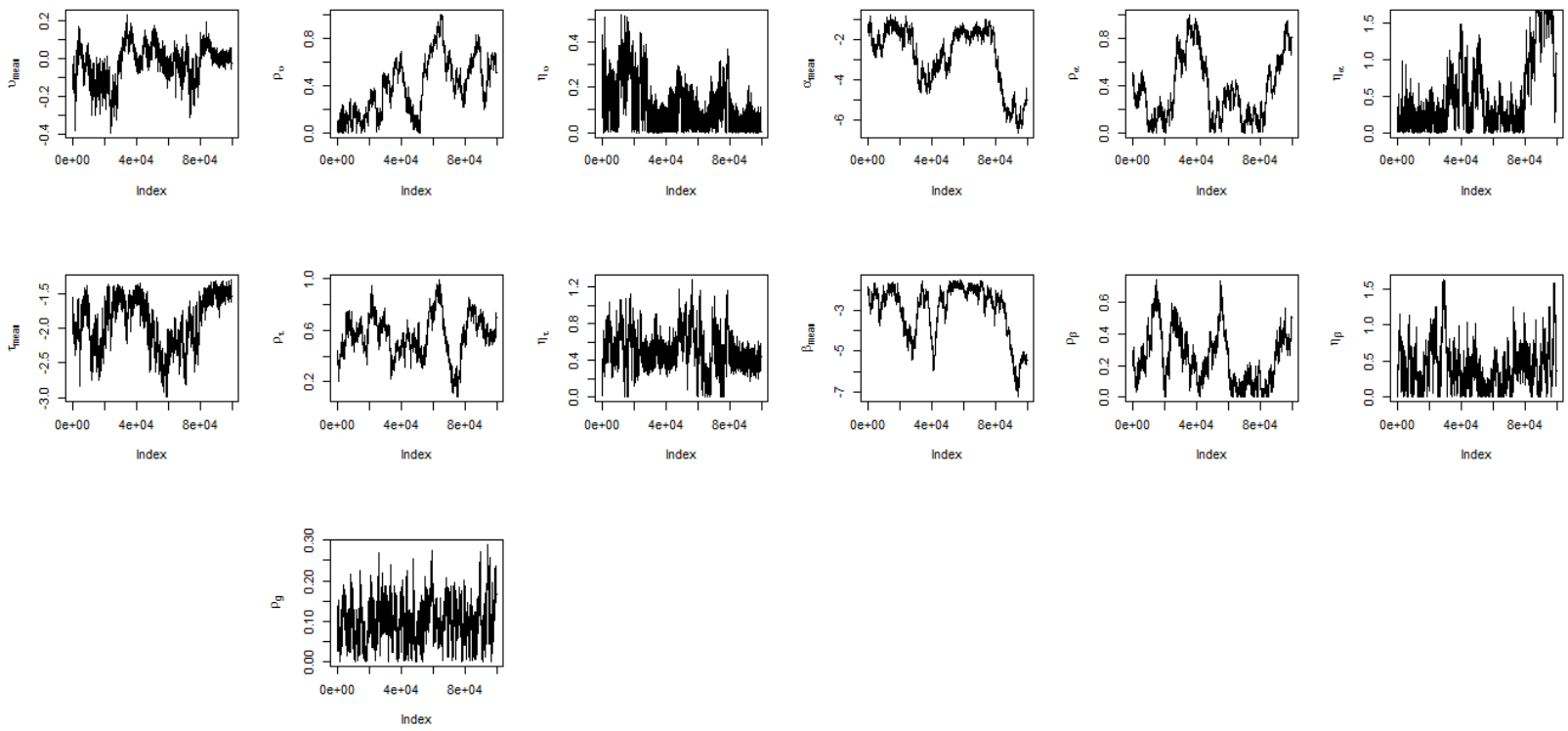


FIGURE 6.9: Trace plots of the mLN parameter estimates for exchange rate data

Table 6.5: Modified Laplace Normal process parameters for Kazakhstan tenge to U.S. dollar exchange rate

Parameter	Mean	95% Interval
$\bar{\upsilon}$	-0.014	(-0.241, 0.136)
ρ_υ	0.464	(0.057, 0.885)
η_υ	0.082	(0.002, 0.295)
$\bar{\tau}$	-1.860	(-2.661, -1.400)
ρ_τ	0.563	(0.192, 0.891)
η_τ	0.486	(0.082, 0.918)
$\bar{\alpha}$	-3.092	(-6.164, -1.268)
ρ_α	0.403	(0.017, 0.915)
η_α	0.593	(0.016, 1.891)
$\bar{\beta}$	-3.165	(-6.171, -1.680)
ρ_β	0.228	(0.009, 0.552)
η_β	0.433	(0.018, 1.325)
ρ_y	0.103	(0.012, 0.217)

The mean and and 95% central intervals for all parameters are shown in Table 6.5. From the results we see that the value of $\bar{\alpha}$ is not significantly different than $\bar{\beta}$, providing little evidence that the changes in the exchange rate do display either positive or negative skewness. All hidden states reflect some measure of stickyness as the mean estimate for the ρ parameters for υ, τ and α are all in the range of 0.40 to 0.50, although there is considerable uncertainty around these estimates. The hidden state υ also does not appear to vary much compared to other state, given η_υ is smaller than the other η parameters.

7

Conclusion

This paper provided a brief introduction into how one current GPU programming language, Thrust, can be used to parallelize the widely used particle filter technique. When using a large number of particles, which allow for better approximations to the likelihood of hidden states in a hidden Markov model, Thrust provides a speed-up of approximately 18 times and 5 times over R and C++ respectively for simple problems. For a more complicated problem, like the time series analysis with multiple hidden states as detailed in Chapter 6, Thrust provides a speed-up of approximately 400 times and 10 times over R and C++, respectively.

These increases, while impressive, are not as high as some of the improvements reported by other researchers as detailed in Chapter 3. This may be partially due to the skill of individual programmer, as some will be better at writing more efficient code than others, and also due to the different problems to which parallel algorithms are being applied. Regardless, the low monetary cost of GPU hardware removes one

barrier to begin experimentation with parallel programming techniques; it will be up to each individual to determine whether the cost to their time is worth the potential benefits.

Bibliography

Aldrich, E. (2013), "GPU Computing in Economics," Working paper.

Baum, L., Petrie, T., Soules, G., and N.Weiss (1970), "A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains." *The annals of mathematical statistics*, pp. 164–171.

Beal, M. and Ghahramani, Z. (2006), "Variational Bayesian learning of directed graphical models with hidden variables," *Bayesian Analysis*, 1, 793–831.

Brun, O., Teuliere, V., and Garcia, J. (2002), "Parallel particle filtering," *Journal of Parallel and Distributed Computing*, 62, 1186–1202.

Crisan, D., Moral, P. D., and Lyons, T. (1998), *Discrete Filtering Using Branching and Interacting Particle Systems*, Univ. Paul Sabatier.

Dempster, A., Laird, N., and Rubin, D. (1977), "Maximum likelihood from incomplete data via the EM algorithm." *Journal of the Royal Statistical Society*, pp. 1–38.

Doucet, A. (2001), *Sequential Monte Carlo Methods*, Wiley Online Library.

Durbin, J. and Koopman, S. (2012), *Time Series Analysis by State Space Methods*, Oxford University Press.

Fernández-Villaverde, J. and Rubio-Ramírez, J. F. (2007), "Estimating macroeconomic models: A likelihood approach," *The Review of Economic Studies*, 74, 1059–1087.

Frühwirth-Schnatter, S. (2006), *Finite Mixture and Markov Switching Models: Modeling and Applications to Random Processes*, Springer.

Gharamani, Z. and Hinton, G. (1996), "Parameter estimation for linear dynamical systems," Tech. Rep. CRG-TR-96-2, University of Toronto, Dept. of Computer Science.

González, A., Roque, A., and García-González, J. (2005), "Modeling and forecasting electricity prices with input/output hidden Markov models," *Power Systems, IEEE Transactions on*, 20, 13–24.

Goodrum, M., Trotter, M., Aksel, A., Acton, S., and Skadron, K. (2012), "Parallelization of particle filter algorithms," in *Computer Architecture*, pp. 139–149, Springer.

Gordon, N. J., Salmond, D. J., and Smith, A. F. (1993), "Novel approach to nonlinear/non-Gaussian Bayesian state estimation," in *IEE Proceedings F (Radar and Signal Processing)*, vol. 140, pp. 107–113, IET.

Hendeby, G., Hol, J., Karlsson, R., and Gustafsson, F. (2007), "A Graphics Processing Unit Implementation of the Particle Filter," Tech. Rep. 2812, Linkping University, The Institute of Technology.

Hendeby, G., Karlsson, R., and Gustafsson, F. (2010), "Particle filtering: the need for speed," *EURASIP Journal on Advances in Signal Processing*, 2010, 22.

Jelinek, F. (1997), *Statistical methods for speech recognition*, MIT Press.

Julier, S. and Uhlmann, J. (1997), "A new extension of the Kalman filter to nonlinear systems," in *Int. symp. aerospace/defense sensing, simul. and controls*, vol. 3, Orlando, FL.

Kalman, R. (1960), "A new approach to linear filtering and prediction problems," *Journal of Basic Engineering*, 82, 35–45.

Karplus, K., Barrett, C., and Hughey, R. (1998), "Hidden Markov models for detecting remote protein homologies." *Bioinformatics*, 14, 846–856.

Krogh, A., Larsson, B., Heijne, G. V., and Sonnhammer, E. (2001), "Predicting transmembrane protein topology with a hidden Markov model: application to complete genomes," *Journal of molecular biology*, 305, 567–580.

Lee, A., Yau, C., Giles, M., Doucet, A., and Holmes, C. (2010), "On the utility of graphics cards to perform massively parallel simulation of advanced Monte Carlo methods," *Journal of Computational and Graphical Statistics*, 19, 769–789.

Murphy, K. (2012), *Machine Learning: A Probabilistic Perspective*, Adaptive computation and machine learning series, MIT Press.

Murray, L. (2012), "GPU acceleration of the particle filter: The Metropolis resampler," *arXiv preprint arXiv:1202.6163*.

Pitt, M. and Shephard, N. (1999), "Filtering via simulation: Auxiliary particle filters," *Journal of the American Statistical Association*, 94, 590–599.

Pole, A., West, M., and Harrison, J. (1994), *Applied Bayesian forecasting and time series analysis*, CRC Press.

Reed, W. J. and Jorgensen, M. (2004), "The double Pareto-lognormal distributiona new parametric model for size distributions," *Communications in Statistics-Theory and Methods*, 33, 1733–1753.

Rydén, T., Teräsvirta, T., and Åsbrink, S. (1998), "Stylized facts of daily return series and the hidden Markov model," *Journal of applied econometrics*, 13, 217–244.

Shumway, R. and Stoffer, D. (2010), *Time Series Analysis and its Applications: with R Examples*, Springer.

Suchard, M., Wang, Q., Chan, C., Frelinger, J., Cron, A., and West, M. (2010), "Understanding GPU programming for statistical computation: Studies in massively parallel massive mixtures," *Journal of Computational and Graphical Statistics*, 19, 419–438.

West, M. (1993), "Mixture models, Monte Carlo, Bayesian updating, and dynamic models," *Computing Science and Statistics*, pp. 325–333.

West, M. and Harrison, J. (1997), *Bayesian Forecasting and Dynamic Models*, Springer Series in Statistics, Springer.